Unit 1: Introduction & Query Processing

Course: PCC-CSE-310G | **Author:** Deepak Modi | **Website:** NotesNeo

Introduction: Architecture, Advantages, Disadvantages, Data models, relational algebra, SQL, Normal forms.
Query Processing: General strategies for query processing, transformations, expected size, statistics in estimation, query improvement. Query evaluation, view processing, query processor.

---

1. Introduction

Database Management Systems (DBMS) are software systems that enable the creation, management, and manipulation of databases. They provide a systematic way to store, retrieve, and manage data efficiently. Advanced DBMS have evolved to handle complex data types, large volumes of data, and high transaction rates while ensuring data integrity and security.

1.1 DBMS Architecture

Database Management System architecture forms the backbone of how data is stored, accessed, and manipulated. Advanced DBMS architectures are designed to handle complex requirements across distributed environments.

1.1.1 Three-Level Architecture

The **ANSI/SPARC model** (also known as the Three-Schema Architecture) is a database architecture standard developed by the American National Standards Institute (ANSI) and the Standards Planning and Requirements Committee (SPARC) in 1975.

What is ANSI/SPARC?

- **ANSI** = American National Standards Institute
- **SPARC** = Standards Planning and Requirements Committee

The ANSI/SPARC model divides database architecture into three abstraction levels:

1. **External Level (View Level)**: This represents how individual users see the data. Different users may have different views of the same database. It allows for multiple user interfaces and hides the complexity of the underlying data structures from end users.

2. **Conceptual Level (Logical Level)**: This represents the logical structure of the entire database as viewed by the Database Administrator (DBA). It defines what data is stored in the database and the relationships among those data. The conceptual schema provides a unified view of the entire database, independent of how data is physically stored.

3. **Internal Level (Physical Level)**: This describes how data is physically stored and accessed on storage devices. It deals with data structures, file organization, and access methods. It provides a low-level view of the database, including how data is stored on disk, indexing methods, and storage allocation.

Diagram: Three-Level Architecture

```
┌──────────────────────┐
│   External Level     │
│    (User Views)      │
└──────────▲───────────┘
           │ Logical Data Independence
┌──────────────────────┐
│   Conceptual Level   │
│ (Logical Structure)  │
└──────────▲───────────┘
           │ Physical Data Independence
┌──────────────────────┐
│   Internal Level     │
│ (Physical Storage)   │
└──────────────────────┘
```

This three-Level architecture provides **data independence**, one of the most valuable features of a DBMS:

- **Logical Data Independence**: The ability to change the conceptual schema without changing external schemas or application programs. Example: Adding a new field to a table without affecting user views or applications.
- **Physical Data Independence**: The ability to change the internal schema without changing the conceptual or external schemas. Example: Upgrading from HDD to SSD storage without affecting how users access data.

Key Benefits:

- **Data Abstraction**: Users can interact with data without needing to understand the complexities of how it is stored.
- **Data Independence**: Changes at one level do not affect other levels, allowing for easier maintenance and evolution of the database.
- **Multiple User Views**: Different users can have different views of the same data, enhancing security and usability.
- **Enhanced Security**: Sensitive data can be hidden from certain users, while still allowing access to necessary information.

1.1.2 Client-Server Architecture

Modern DBMS often follow a client-server architecture, where the database system is divided into two main components:

- **Client**: The user interface or application that interacts with the database. It sends requests to the server and displays results to the user.
- **Server**: The database engine that processes requests, manages data storage, and performs operations on the database.

```
┌─────────────────┐       ┌──────────────────┐
│     Client      │       │      Server      │
│(User Interface) │◄─────►│(Database Engine) │
└─────────────────┘       └──────────────────┘
```

This architecture allows for centralized data management while enabling multiple clients to access the database concurrently.

1.1.3 Distributed DBMS Architecture

In advanced systems, data may be distributed across multiple locations to improve performance, availability, and fault tolerance. A distributed DBMS allows data to be stored on multiple database nodes, which can be located in different geographical locations.

```
┌──────────────────┐     ┌──────────────────┐     ┌──────────────────┐
│  Database Node   │     │  Database Node   │     │  Database Node   │
│    (Site A)      │◄───►│    (Site B)      │◄───►│    (Site C)      │
└──────────────────┘     └──────────────────┘     └──────────────────┘
         │                        │                        │
         │                        │                        │
         └────────────────────────┼────────────────────────┘
                                  │
                                  │
                      ┌───────────▼───────────┐
                      │   Global Directory    │
                      │   & Query Processor   │
                      └───────────────────────┘
```

This distribution enables:

- **Data replication**: For fault tolerance and faster local access
- **Data partitioning**: For improved performance and scalability
- **Location transparency**: Users need not know where data physically resides

1.2 Advantages & Disadvantages of Advanced DBMS

1.2.1 Advantages

1. Enhanced Data Sharing and Integration: Advanced DBMS systems excel at integrating disparate data sources, enabling organizations to create unified views of their data. This facilitates better decision-making through comprehensive analytics.

2. Improved Security and Compliance: Modern DBMS implement sophisticated security mechanisms including role-based access control, encryption, and auditing features that help organizations meet regulatory requirements like GDPR or HIPAA.

3. Better Performance and Scalability: Advanced DBMS can handle massive datasets and concurrent users through techniques like query optimization, indexing strategies, and parallel processing. Cloud-based DBMS solutions offer virtually unlimited scalability.

4. Increased Availability and Reliability: Features like replication, failover clustering, and automated backup/recovery ensure that data remains accessible even during hardware failures or maintenance windows.

5. Support for Complex Data Types: Beyond traditional relational data, advanced DBMS can now efficiently store and process spatial data, time series, documents, graphs, and other specialized data formats.

1.2.2 Disadvantages

1. Increased Complexity: Advanced DBMS features come with greater complexity in setup, administration, and optimization. Organizations often need specialized database administrators (DBAs).

2. Higher Costs: Enterprise-grade DBMS solutions involve significant costs in terms of licensing, hardware requirements, and skilled personnel.

3. Performance Overhead: Some advanced features like extensive logging, encryption, or complex integrity constraints can introduce performance overhead.

4. Lock-in Concerns: Organizations may become dependent on proprietary features of a specific DBMS vendor, making it difficult to migrate to alternative solutions.

5. Learning Curve: Developers and administrators need continuous learning to keep up with rapidly evolving DBMS technologies and best practices.

1.3 Data Models

Data models define how data is structured, stored, and manipulated in a database. They provide a conceptual framework for organizing data and establishing relationships between different data entities.

Different types of data models:

1. Relational Data Model
2. Object-Oriented Data Model
3. Object-Relational Data Model

4. Entity-Relationship Model (ER Model)
5. Hierarchical Data Model
6. Network Data Model
7. NoSQL Data Models (Document, Key-Value, Column-Family, Graph)

1.3.1 Relational Data Model

The relational model organizes data into tables (relations) consisting of rows (tuples) and columns (attributes). It remains the most widely used data model due to its simplicity and proven reliability.

Row is also called tuple or record, and column is also called attribute or field. Each table has a unique primary key that identifies each row, and relationships between tables are established through foreign keys.

Key characteristics:

- Data organized in tables with rows and columns
- Each table has a unique primary key
- Relationships established through foreign keys
- Strong mathematical foundation based on set theory
- ACID (Atomicity, Consistency, Isolation, Durability) properties

Example:

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id)
);
```

1.3.2 Object-Oriented Data Model

The object-oriented model extends database capabilities by integrating object-oriented programming concepts. It allows for complex data types and relationships, making it suitable for applications requiring rich data representation.

Key characteristics:

- Encapsulation of data and methods
- Support for complex data types
- Inheritance and polymorphism
- Object identity independent of values

Example:

```
CREATE TYPE Address AS OBJECT (
    street VARCHAR(100),
    city VARCHAR(50),
    state CHAR(2),
    zip VARCHAR(10)
);

CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    home_address Address,
    work_address Address
);
```

1.3.3 Object-Relational Data Model

The object-relational model is hybrid model which combines features of both relational and object-oriented models. It allows for the definition of complex data types while maintaining the relational model's principles. This model is particularly useful for applications that require both structured and unstructured data handling.

Key characteristics:

- Supports user-defined types and complex objects
- Maintains relational foundations
- Provides SQL language extensions
- Enables references and inheritance

1.3.4 Entity-Relationship Model (ER Model)

The Entity-Relationship (ER) model is a high-level conceptual data model used to describe the structure of a database. It represents data as entities, attributes, and relationships, providing a visual representation of how different data elements are related.

Key components:

- **Entity**: A real-world object or concept that can have data stored about it (e.g., Student, Course). Represented as rectangles in ER diagrams.
- **Attribute**: A property or characteristic of an entity (e.g., Student Name, Course Code). Represented as ovals connected to entities.
- **Relationship**: An association between two or more entities (e.g., Student enrolls in Course). Represented as diamonds connecting entities.

Example ER Diagram:

```
[Student] --enrolls--> [Course]
  |                      |
  |-- has name           |-- has code
```

ER diagrams are often used during the database design phase to visualize the structure before implementation. They can be converted into relational tables for actual database creation.

1.3.5 Hierarchical Data Model

The hierarchical data model organizes data in a tree-like structure, where each record has a single parent and can have multiple children. This model is suitable for representing one-to-many relationships.

Key characteristics:

- Data organized in a tree structure
- Each record has a single parent
- Supports one-to-many relationships
- Simple and efficient for certain applications

Example:

```
CREATE TABLE Departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100),
    parent_department_id INT,
    FOREIGN KEY (parent_department_id) REFERENCES Departments(department_id)
);
```

In this example, departments can have sub-departments, forming a hierarchical structure where each department can have only one parent department.

1.3.6 Network Data Model

The network data model is an extension of the hierarchical model, allowing more complex relationships between records. It organizes data in a graph structure, where entities can have multiple relationships with each other.

Key characteristics:

- Data represented as records connected by links
- Supports many-to-many relationships
- More flexible than hierarchical model but more complex

Example:

```
CREATE TYPE Employee AS OBJECT (
    emp_id INT,
    name VARCHAR(100),
    department_id INT,
    manager_id INT
);

CREATE TABLE Employees OF Employee;
```

In this example, an employee can have multiple managers or be part of multiple departments, allowing for more complex relationships than traditional hierarchical models.

1.3.7 NoSQL Data Models

NoSQL ("Not Only SQL") data models are designed to handle large volumes of unstructured or semi-structured data. They provide flexibility and scalability, making them suitable for big data applications and real-time analytics.

NoSQL data models address specific needs that relational databases struggle with, particularly around scalability, flexibility, and handling unstructured data. NoSQL databases can be categorized into several types, each optimized for different use cases.

Document Model:

- Stores data as flexible, semi-structured documents (often JSON)
- Ideal for applications with varying data structures
- Supports nested data and arrays
- Example databases: MongoDB, CouchDB

```
// MongoDB example
{
  "_id": 1001,
  "name": "Deepak Modi",
  "courses": [
    { "code": "CS101", "grade": "A" },
    { "code": "MATH202", "grade": "B+" }
  ],
  "address": {
    "street": "123 Campus Road",
    "city": "University Town"
  }
}
```

Key-Value Model:

- Stores data as key-value pairs, where each key is unique

- Highly scalable and fast for simple lookups
- Suitable for caching and session management
- Example databases: Redis, Amazon DynamoDB

```
// Redis example
SET user:1001:name "Deepak Modi"
SET user:1001:email "Deepak.Modi@example.com"
```

Column-Family Model:

- Stores data in column families, where each row can have a different set of columns
- Optimized for read and write performance on large datasets
- Suitable for analytical applications and time-series data
- Example databases: Apache Cassandra, HBase

```
// Cassandra example
CREATE TABLE user_profiles (
    user_id UUID PRIMARY KEY,
    name TEXT,
    email TEXT,
    address TEXT,
    preferences MAP<TEXT, TEXT>
);
```

Graph Model:

- Represents data as nodes (entities) and edges (relationships)
- Optimized for traversing complex relationships
- Ideal for social networks, recommendation systems, and network analysis
- Example databases: Neo4j, Amazon Neptune

```
// Neo4j example
MATCH (a:Person)-[r:FRIENDS_WITH]->(b:Person)
WHERE a.name = "Deepak Modi"
RETURN b.name
```

1.4 Relational Algebra

Relational algebra is a procedural query language used to manipulate and retrieve data from relational databases. It provides a set of operations that can be applied to relations (tables) to produce new relations. Relational algebra forms the theoretical foundation for SQL and other query languages.

Relational algebra operations can be classified into basic operations, derived operations, and extended operations.

1.4.1 Basic Operations

1. Selection (σ): Selects tuples (rows) from a relation that satisfy a given condition.

Syntax: $\sigma_{\text{condition}}$(Relation)

Example: $\sigma_{\text{salary} > 50000}$(Employee) - Returns all employees with salary greater than 50000.

Consider this Employee relation:

EmpID	Name	Salary	Department
101	Deepak	45000	IT
102	Nitish	65000	HR
103	Vishal	52000	Finance
104	Prateek	48000	IT

The operation $\sigma_{\text{salary} > 50000}$(Employee) returns:

EmpID	Name	Salary	Department
102	Nitish	65000	HR
103	Vishal	52000	Finance

2. Projection (π): Extracts specific columns (attributes) from a relation.

Syntax: $\pi_{\text{attributes}}$(Relation)

Example: $\pi_{\text{name, department}}$(Employee) - Returns only name and department columns.

Using the same Employee relation, $\pi_{\text{name, department}}$(Employee) returns:

Name	Department
Deepak	IT
Nitish	HR
Vishal	Finance
Prateek	IT

3. Union (∪): Combines tuples (rows) from two relations, removing duplicates.

Syntax: R ∪ S (Both relations must be union-compatible)

Consider these two relations:

Department1:

DeptID	DeptName
D1	IT
D2	HR

Department2:

DeptID	DeptName
D2	HR
D3	Finance

The operation Department1 ∪ Department2 returns:

DeptID	DeptName
D1	IT
D2	HR
D3	Finance

4. Set Difference (−): Returns tuples that exist in the first relation but not in the second.

Syntax: R − S

Using the Department1 and Department2 relations:

Department1 − Department2 returns:

DeptID	DeptName
D1	IT

5. Cartesian Product (×): Combines each tuple of the first relation with every tuple of the second.

Syntax: R × S

Consider these relations:

Students:

SID	Name
1	Ritvik
2	Ranjit

Courses:

CID	Course
C1	Math
C2	CS

Students × Courses returns:

SID	Name	CID	Course
1	Ritvik	C1	Math
1	Ritvik	C2	CS
2	Ranjit	C1	Math
2	Ranjit	C2	CS

6. Rename (ρ): Renames a relation or its attributes.

Syntax: ρ_{new_name}(Relation)

Example: $\rho_{Worker(ID,Name,Pay,Dept)}$(Employee)

1.4.2 Derived Operations

1. Intersection (∩): Returns tuples that are present in both relations.

Syntax: R ∩ S

Using the Department1 and Department2 relations:

Department1 ∩ Department2 returns:

DeptID	DeptName
D2	HR

2. Join (⋈): Combines related tuples from two relations based on a join condition.

Natural Join: Combines tuples that have the same values for common attributes.

Consider:

Employee:

EmpID	Name	DeptID
101	Deepak	D1
102	Nitish	D2
103	Vishal	D3

Department:

DeptID	DeptName	Location
D1	IT	Floor 1
D2	HR	Floor 2
D4	Admin	Floor 3

Employee ⋈ Department (Natural Join) returns:

EmpID	Name	DeptID	DeptName	Location
101	Deepak	D1	IT	Floor 1
102	Nitish	D2	HR	Floor 2

Theta Join: Combines tuples that satisfy a specified condition.

Employee $\bowtie_{\text{Employee.DeptID} < \text{Department.DeptID}}$ Department returns:

EmpID	Name	DeptID	DeptID	DeptName	Location
101	Deepak	D1	D2	HR	Floor 2
101	Deepak	D1	D4	Admin	Floor 3
102	Nitish	D2	D4	Admin	Floor 3

3. Division (÷): Returns tuples from the first relation that are associated with all tuples in the second relation.

Consider:

EmployeeSkills:

EmpID	Skill
101	Java
101	Python
102	Java
103	Python
103	SQL

SkillsRequired:

Skill
Java
Python

EmployeeSkills ÷ SkillsRequired returns:

EmpID
101

This indicates that Employee with ID 101 has all required skills.

1.4.3 Extended Operations

1. Outer Joins: Outer joins preserve tuples from one or both relations even if there is no match, extending regular joins to include non-matching tuples.

There are three types of outer joins:

- **Left Outer Join (⟕)**: Preserves all tuples from the left relation, filling missing values with NULL
- **Right Outer Join (⟖)**: Preserves all tuples from the right relation, filling missing values with NULL
- **Full Outer Join (⟗)**: Preserves all tuples from both relations, filling missing values with NULL

Using the Employee and Department relations from above:

Left Outer Join (Employee ⟕ Department):

EmpID	Name	DeptID	DeptName	Location
101	Deepak	D1	IT	Floor 1
102	Nitish	D2	HR	Floor 2

EmpID	Name	DeptID	DeptName	Location
103	Vishal	D3	NULL	NULL

Right Outer Join (Employee ⟖ Department):

EmpID	Name	DeptID	DeptName	Location
101	Deepak	D1	IT	Floor 1
102	Nitish	D2	HR	Floor 2
NULL	NULL	D4	Admin	Floor 3

Full Outer Join (Employee ⟗ Department):

EmpID	Name	DeptID	DeptName	Location
101	Deepak	D1	IT	Floor 1
102	Nitish	D2	HR	Floor 2
103	Vishal	D3	NULL	NULL
NULL	NULL	D4	Admin	Floor 3

2. Aggregation Operations: Functions like SUM, AVG, MAX, MIN, and COUNT applied to groups of tuples.

Consider:

Sales:

Product	Region	Amount
Laptop	North	2000
Laptop	South	1500
Tablet	North	800
Phone	South	500
Phone	North	600

$G_{\text{Product, SUM(Amount)}}$(Sales):

Product	SUM(Amount)
Laptop	3500

Product	SUM(Amount)
Tablet	800
Phone	1100

3. Recursive Closure: Used for queries involving hierarchical or graph-like data structures.

Example: Finding all direct and indirect reports in a management hierarchy.

Consider:

Manager:

Employee	Manager
Vishal	Nitish
Prateek	Nitish
Sandeep	Vishal
Sumit	Prateek
Ankit	Sandeep

Using recursive closure, we can find all direct and indirect reports of Nitish:

Employee
Vishal
Prateek
Sandeep
Sumit
Ankit

1.5 Structured Query Language (SQL)

SQL (Structured Query Language) is the standard proceduaral language for managing and manipulating relational databases. It provides a powerful and flexible way to interact with data, allowing users to perform operations such as querying, updating, inserting, and deleting data.

SQL is a declarative language, meaning users specify what they want to achieve without detailing how to do it. It is widely used across various database systems, including MySQL, PostgreSQL, Oracle, SQL Server, and SQLite.

1.5.1 SQL Categories

SQL commands are organized into five main categories: Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), Data Control Language (DCL) and Transaction Control Language (TCL).

```
                    ┌─────────────────────┐
                    │    SQL Commands     │
                    └──────────┬──────────┘
                               │
    ┌──────────┬──────────┬────┴────┬──────────┐
    │          │          │         │          │
┌───▼────┐ ┌───▼────┐ ┌───▼────┐ ┌──▼───┐  ┌───▼────┐
│  DDL   │ │  DML   │ │  DQL   │ │ DCL  │  │  TCL   │
└───┬────┘ └───┬────┘ └───┬────┘ └──┬───┘  └───┬────┘
    │          │          │         │          │
    │          │          │         │          │
┌───▼────┐ ┌───▼────┐ ┌───▼────┐ ┌──▼───┐  ┌───▼──────────────┐
│ CREATE │ │ INSERT │ │ SELECT │ │GRANT │  │     COMMIT      │
│ ALTER  │ │ UPDATE │ │        │ │REVOKE│  │    ROLLBACK     │
│ DROP   │ │ DELETE │ │        │ │      │  │    SAVEPOINT    │
│TRUNCATE│ │ MERGE  │ │        │ │      │  │ SET TRANSACTION │
└────────┘ └────────┘ └────────┘ └──────┘  └─────────────────┘
```

1. Data Definition Language (DDL)

- Used to **define or change the structure** of database objects like tables, schemas, etc.
- These commands **create, modify, or delete** tables and other objects.

Examples:

- `CREATE` – To create a new table or database.
- `ALTER` – To modify an existing table.
- `DROP` – To delete a table or database.
- `TRUNCATE` – To remove all records from a table, but keep the structure.

2. Data Manipulation Language (DML)

- Used to **manipulate data stored** in the tables.
- These commands **insert, update, delete, or retrieve** data.

Examples:

- `INSERT` – To add new data into a table.
- `UPDATE` – To modify existing data.
- `DELETE` – To remove data.
- `SELECT` – To fetch data (sometimes considered as DQL – see below).

3. Data Query Language (DQL)

- Focuses only on **querying or fetching data** from the database.

Example:

- `SELECT` – Used to retrieve data from one or more tables.

> Note: Some books count `SELECT` under DML, but many treat it separately as DQL.

4. Data Control Language (DCL)

- Deals with **permissions and access control** in the database.

Examples:

- `GRANT` – Give access rights to users.
- `REVOKE` – Remove access rights from users.

5. Transaction Control Language (TCL)

- Manages **transactions** in the database (group of DML operations).
- Ensures **data consistency and rollback if needed** during errors.

Examples:

- `COMMIT` – Save changes permanently to the database.
- `ROLLBACK` – Undo changes made during the transaction.
- `SAVEPOINT` – Set a point to which you can roll back.
- `SET TRANSACTION` – Define properties of the transaction.

1.5.2 SQL Command Examples with Results

Let's take a complete example using a Student database to demonstrate all SQL categories:

1. DDL Example - Creating and Modifying Structure

```sql
-- Create a new table
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    major VARCHAR(50),
    gpa DECIMAL(3,2)
);

-- Alter the table to add a new column
ALTER TABLE Students ADD email VARCHAR(100);
```

```
-- (Later) Drop the table when no longer needed
-- DROP TABLE Students;
```

Result: Table structure created in database

```
mysql> DESCRIBE Students;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| student_id | int          | NO   | PRI | NULL    |       |
| name       | varchar(100) | NO   |     | NULL    |       |
| age        | int          | YES  |     | NULL    |       |
| major      | varchar(50)  | YES  |     | NULL    |       |
| gpa        | decimal(3,2) | YES  |     | NULL    |       |
| email      | varchar(100) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
```

2. DML Example - Manipulating Data

```
-- Insert data
INSERT INTO Students (student_id, name, age, major, gpa, email) VALUES
(101, 'Deepak Modi', 21, 'Computer Science', 3.85, 'deepak@example.com'),
(102, 'Nitish Kumar', 22, 'Engineering', 3.67, 'nitish@example.com'),
(103, 'Vishal Singh', 20, 'Mathematics', 3.92, 'vishal@example.com'),
(104, 'Prateek Shah', 23, 'Computer Science', 3.45, 'prateek@example.com');

-- Update data
UPDATE Students
SET gpa = 3.95
WHERE student_id = 103;

-- Delete data
DELETE FROM Students
WHERE student_id = 104;
```

Result after operations:

```
mysql> SELECT * FROM Students;
+------------+--------------+------+------------------+------+--------------------+
| student_id | name         | age  | major            | gpa  | email              |
+------------+--------------+------+------------------+------+--------------------+
| 101        | Deepak Modi  | 21   | Computer Science | 3.85 | deepak@example.com |
| 102        | Nitish Kumar | 22   | Engineering      | 3.67 | nitish@example.com |
| 103        | Vishal Singh | 20   | Mathematics      | 3.95 | vishal@example.com |
+------------+--------------+------+------------------+------+--------------------+
```

3. DQL Example - Querying Data

```sql
-- Basic select
SELECT * FROM Students WHERE gpa > 3.8;

-- Aggregation and grouping
SELECT major, AVG(gpa) as average_gpa
FROM Students
GROUP BY major;

-- Ordering results
SELECT name, major, gpa
FROM Students
ORDER BY gpa DESC;
```

Result of first query:

```
mysql> SELECT * FROM Students WHERE gpa > 3.8;
+------------+--------------+------+------------------+------+--------------------+
| student_id | name         | age  | major            | gpa  | email              |
+------------+--------------+------+------------------+------+--------------------+
| 101        | Deepak Modi  | 21   | Computer Science | 3.85 | deepak@example.com |
| 103        | Vishal Singh | 20   | Mathematics      | 3.95 | vishal@example.com |
+------------+--------------+------+------------------+------+--------------------+
```

4. DCL Example - Access Control

```sql
-- Create a new user
CREATE USER 'student_readonly'@'localhost' IDENTIFIED BY 'password123';

-- Grant read permissions
GRANT SELECT ON university.Students TO 'student_readonly'@'localhost';

-- Revoke permissions
REVOKE SELECT ON university.Students FROM 'student_readonly'@'localhost';
```

Result:

```
mysql> SHOW GRANTS FOR 'student_readonly'@'localhost';
+-------------------------------------------------------------------------+
| Grants for student_readonly@localhost                                   |
+-------------------------------------------------------------------------+
| GRANT SELECT ON university.Students TO `student_readonly`@`localhost`  |
+-------------------------------------------------------------------------+
```

5. TCL Example - Transaction Control

```sql
-- Start a transaction
START TRANSACTION;

-- Make some changes
UPDATE Students SET gpa = gpa + 0.1 WHERE major = 'Computer Science';
INSERT INTO Students VALUES (104, 'Sumit Verma', 22, 'Physics', 3.75, 'sumit@example.com');

-- Check intermediate state (only visible in this session)
SELECT * FROM Students;

-- Decide to roll back changes
ROLLBACK;

-- Or alternatively, to commit changes:
-- COMMIT;
```

Result during transaction:

```
mysql> SELECT * FROM Students;
+------------+-------------+------+------------------+------+--------------------+
| student_id | name        | age  | major            | gpa  | email              |
+------------+-------------+------+------------------+------+--------------------+
| 101        | Deepak Modi | 21   | Computer Science | 3.95 | deepak@example.com |
| 102        | Nitish Kumar | 22  | Engineering      | 3.67 | nitish@example.com |
| 103        | Vishal Singh | 20  | Mathematics      | 3.95 | vishal@example.com |
| 104        | Sumit Verma | 22   | Physics          | 3.75 | sumit@example.com  |
+------------+-------------+------+------------------+------+--------------------+
```

Result after ROLLBACK:

```
mysql> SELECT * FROM Students;
+------------+-------------+------+------------------+------+--------------------+
| student_id | name        | age  | major            | gpa  | email              |
+------------+-------------+------+------------------+------+--------------------+
| 101        | Deepak Modi | 21   | Computer Science | 3.85 | deepak@example.com |
| 102        | Nitish Kumar | 22  | Engineering      | 3.67 | nitish@example.com |
| 103        | Vishal Singh | 20  | Mathematics      | 3.95 | vishal@example.com |
+------------+-------------+------+------------------+------+--------------------+
```

1.5.3 Advanced SQL Features

Modern SQL implementations offer powerful features beyond basic CRUD operations.

1. Subqueries (Nested Queries) Subqueries are SQL queries embedded within another query:

```sql
-- Find students with GPA higher than average
SELECT name, gpa FROM Students
```

```sql
WHERE gpa > (SELECT AVG(gpa) FROM Students);
```

2. Common Table Expressions (CTEs) CTEs improve query readability and enable recursive queries:

```sql
-- Calculate average GPA by major
WITH MajorAvgs AS (
    SELECT major, AVG(gpa) AS avg_gpa
    FROM Students GROUP BY major
)
SELECT major, avg_gpa FROM MajorAvgs
WHERE avg_gpa > 3.5;
```

3. Window Functions Window functions perform calculations across rows related to the current row:

```sql
-- Rank students by GPA within each major
SELECT name, major, gpa,
       RANK() OVER (PARTITION BY major ORDER BY gpa DESC) as rank
FROM Students;
```

4. Stored Procedures and Functions Database objects containing SQL statements for reuse:

```sql
CREATE PROCEDURE UpdateStudentGPA(student_id INT, new_gpa DECIMAL(3,2))
BEGIN
    UPDATE Students SET gpa = new_gpa WHERE student_id = student_id;
END;
```

1.6 Normal Forms (1NF to BCNF)

Normalization is the process of organizing data in database to minimize redundancy and dependency. The goal is to reduce data anomalies and improve data integrity. It involves dividing large tables into smaller ones and defining relationships between them.

Normalization is typically done in stages, known as normal forms. Each normal form has specific rules and requirements that must be met.

Normal Form	Description	Requirements
1NF	First Normal Form	Atomic values, unique rows, no repeating groups
2NF	Second Normal Form	1NF + No partial dependency on composite keys
3NF	Third Normal Form	2NF + No transitive dependency
BCNF	Boyce-Codd Normal Form	3NF + Every functional dependency has a superkey on the left side

Normal Form	Description	Requirements
4NF	Fourth Normal Form	BCNF + No multi-valued dependencies
5NF	Fifth Normal Form	4NF + No join dependencies

1.6.1 First Normal Form (1NF)

A relation is in 1NF if:

- Each attribute (column) contains only atomic (indivisible) values
- Each row is unique (no duplicate rows) and can be identified by a primary key
- No repeating groups or arrays

Example: ❌ Not in 1NF:

```
| StudentID | Name | Subjects |
|-----------|------|----------|
| 1         | Aman | DBMS, OS |
| 2         | Ravi | Networks |
```

Here, the `Subjects` column has **multiple values** in a single cell → violates 1NF ❌

✅ After 1NF:

```
| StudentID | Name | Subject  |
|-----------|------|----------|
| 1         | Aman | DBMS     |
| 1         | Aman | OS       |
| 2         | Ravi | Networks |
```

Now each cell contains only atomic values, achieving 1NF.

1.6.2 Second Normal Form (2NF)

A relation is in 2NF if:

- It is in 1NF
- No partial dependency (i.e., non-key attributes must be fully functionally dependent on the entire primary key)

This is especially important for tables with composite primary keys (more than one column as the key).

> Partial dependency means a non-key attribute (column) depends on only part of a composite primary key (not the whole key). 2NF eliminates this by ensuring all non-key attributes depend on the complete primary key not just part of it.

Example:
❌ **Not in 2NF:** Composite Primary Key = (StudentID, Subject)

```
| StudentID | Subject  | StudentName |
|-----------|----------|-------------|
| 1         | DBMS     | Aman        |
| 1         | OS       | Aman        |
| 2         | Networks | Ravi        |
```

Here, `StudentName` depends **only on StudentID**, not on the full composite key (StudentID, Subject) → **partial dependency** → violates 2NF ❌

✅ **After 2NF:** Split into two tables:

```
Table: Students
| StudentID | StudentName |
|-----------|-------------|
| 1         | Aman        |
| 2         | Ravi        |

Table: Enrollment
| StudentID | Subject  |
|-----------|----------|
| 1         | DBMS     |
| 1         | OS       |
| 2         | Networks |
```

By achieving 2NF, we ensure that non-key attributes are fully dependent on the primary key, eliminating redundancy and potential anomalies.

1.6.3 Third Normal Form (3NF)

A relation is in 3NF if:

- It is in 2NF
- No transitive dependency (i.e., non-key attributes must not depend on other non-key attributes)

> Transitive dependency means that if A → B and B → C, then A → C. In 3NF, we want to eliminate such dependencies.

Example:
❌ **Not in 3NF:**

```
| EmpID | DeptID | DeptName |
|-------|--------|----------|
| 101   | D1     | HR       |
| 102   | D2     | Finance  |
```

Here, `DeptName` depends on `DeptID`, not directly on `EmpID` → **transitive dependency** → violates 3NF ❌

✅ **After 3NF:** Split into two tables:

```
Table: Employee
| EmpID | DeptID |
|-------|--------|
| 101   | D1     |
| 102   | D2     |

Table: Department
| DeptID | DeptName |
|--------|----------|
| D1     | HR       |
| D2     | Finance  |
```

1.6.4 Boyce-Codd Normal Form (BCNF)

BCNF is a stricter version of 3NF. A relation is in BCNF if:

- It is in 3NF
- For every functional dependency X → Y, X must be a superkey of the relation

> This means that every **determinant** (left side of functional dependency) must be a **superkey**.

Example: ❌ Not in BCNF:

```
| Course | Instructor | Room     |
|--------|------------|----------|
| DBMS   | Prof. A    | Room 101 |
| OS     | Prof. B    | Room 102 |
```

Functional Dependencies:

- Course → Instructor
- Instructor → Room

Here, `Instructor` is not a superkey but determines `Room` → violates BCNF ❌

✅ **After BCNF:** Split into two tables:

```
Table: Course
| Course | Instructor |
|--------|------------|
| DBMS   | Prof. A    |
| OS     | Prof. B    |

Table: Instructor
| Instructor | Room     |
|------------|----------|
| Prof. A    | Room 101 |
| Prof. B    | Room 102 |
```

Normalizing to BCNF eliminates certain anomalies that might still exist in 3NF relations.

1.6.5 Fourth Normal Form (4NF)

A relation is in 4NF if:

- It is in BCNF
- There are no multi-valued dependencies (MVDs)

> Multi-valued dependency means that one attribute can have multiple independent values for a single key.

Example: ❌ Not in 4NF:

```
| Student | Hobby   | Language |
|---------|---------|----------|
| Ram     | Cricket | English  |
| Ram     | Music   | English  |
| Ram     | Cricket | Hindi    |
| Ram     | Music   | Hindi    |
```

Here, both `Hobby` and `Language` are **independent** of each other but both depend on `Student`, leading to redundancy → violates 4NF ❌

✅ **After 4NF:** Split into two tables:

```
Table: StudentHobbies
| Student | Hobby   |
|---------|---------|
| Ram     | Cricket |
| Ram     | Music   |

Table: StudentLanguages
| Student | Language |
|---------|----------|
| Ram      | English  |
| Ram      | Hindi    |
```

1.6.6 Fifth Normal Form (5NF)

A relation is in 5NF if:

- It is in 4NF
- Every join dependency in the relation is implied by the candidate keys

> This means that all data can be reconstructed from smaller pieces without losing information.

5NF deals with cases where information can be derived from other tables, ensuring no redundancy.

Example: ❌ Not in 5NF:

```
| Project | Team   | Tool  |
|---------|--------|-------|
| P1      | Team A | Figma |
| P1      | Team A | Canva |
| P1      | Team B | Figma |
| P1      | Team B | Canva |
```

All combinations are stored, leading to redundancy → violates 5NF ❌

✅ **After 5NF:** Split into three tables:

```
Table: ProjectTeam
| Project | Team   |
|---------|--------|
| P1      | Team A |
| P1      | Team B |

Table: ProjectTool
| Project | Tool  |
|---------|-------|
| P1      | Figma |
| P1      | Canva |

Table: TeamTool (if needed)
| Team   | Tool  |
|--------|-------|
| Team A | Figma |
| Team A | Canva |
| Team B | Figma |
| Team B | Canva |
```

This ensures that each piece of information is stored only once, achieving 5NF.

2. Query Processing

Query processing is the process of converting high-level SQL queries into low-level instructions that the database engine can understand and execute. It involves analyzing, optimizing, and executing the query to fetch the result as quickly as possible.

2.1 General Strategies for Query Processing

Query processing follows a multi-step approach to transform user queries into efficient execution plans. The main goal is to minimize the cost of executing the query while ensuring correctness and completeness.

2.1.1 Query Processing Phases

1. **Query Parsing and Translation**
    - Parse the SQL query to check syntax
    - Translate the query into an internal representation (often relational algebra)
    - Validate the query against schema metadata
2. **Query Optimization**
    - Generate alternative execution plans
    - Estimate the cost of each plan
    - Select the plan with the lowest estimated cost
3. **Query Execution**
    - Execute the chosen plan
    - Fetch data from storage
    - Process intermediate results
    - Return the final result to the user
4. **Result Delivery**
    - Format results according to user requirements
    - Return the results to the client application

2.1.2 Basic Processing Strategies

The two main strategies for query processing are materialization and pipelining.

Materialization Strategy

- Fully computes intermediate results
- Stores temporary results before proceeding to the next operation

- Better for small or moderate datasets

Pipelining Strategy

- Passes tuples to the next operation without storing intermediate results
- Reduces I/O and storage costs
- Suitable for stream processing and large datasets

Hybrid Approaches

- Combine materialization and pipelining based on query characteristics
- Use materialization for critical operations and pipelining for others

Materialization vs Pipelining

Feature	Materialization	Pipelining
Intermediate Results	Stored in memory or disk	Passed directly to next step
I/O Cost	Higher due to storage	Lower due to no intermediate storage
Memory Usage	Higher due to temp storage	Lower as no temp storage needed
Suitability	Small/Moderate datasets	Large datasets or streams
Processing Overhead	More overhead due to temp storage	Less overhead, more efficient

2.2 Query Transformations

Query transformation is the process of rewriting a query into an equivalent form that is more efficient to execute without changing the result. This is a crucial step in query optimization, as it can significantly impact the performance of the query execution.

Goals of Query Transformation:

- To improve performance.
- To reduce computation cost (less time, CPU, I/O).
- To help the optimizer generate a better query plan.

Transformations can be broadly classified into two categories: algebraic transformations and semantic transformations.

2.2.1 Algebraic Transformations

These transformations are based on the properties of relational algebra operations and aim to rewrite queries into equivalent but more efficient forms.

1. Selection Push-Down

Push selection operations as close as possible to the data sources to reduce intermediate result sizes.

Original: $\sigma_{\text{condition}}(R \bowtie S)$

Transformed: $\sigma_{\text{condition on R}}(R) \bowtie \sigma_{\text{condition on S}}(S)$

Example: Original:

```sql
SELECT *
FROM Orders O JOIN Customers C ON O.customer_id = C.customer_id
WHERE O.order_date > '2023-01-01' AND C.region = 'North';
```

Transformed:

```sql
SELECT *
FROM (SELECT * FROM Orders WHERE order_date > '2023-01-01') O
JOIN (SELECT * FROM Customers WHERE region = 'North') C
ON O.customer_id = C.customer_id;
```

2. Projection Push-Down

Push projections down to eliminate unnecessary columns early in processing.

Original: $\pi_{\text{attributes}}(R \bowtie S)$

Transformed: $\pi_{\text{final attributes}}(\pi_{\text{attributes from R + join attributes}}(R) \bowtie \pi_{\text{attributes from S + join attributes}}(S))$

3. Join Reordering

Rearrange the order of joins to minimize intermediate results.

For joins $R \bowtie S \bowtie T$, there are different possible execution orders:

- $(R \bowtie S) \bowtie T$
- $R \bowtie (S \bowtie T)$

4. Common Subexpression Elimination

Identify and compute repeated subexpressions only once.

2.2.2 Semantic Transformations

These transformations use semantic information like constraints and dependencies to simplify queries.

1. Predicate Simplification

Simplify complex conditions based on known constraints.

2. Redundant Join Elimination

Remove unnecessary joins when they don't contribute to the result.

3. View Merging

Replace view references with their definitions for comprehensive optimization.

4. Subquery Flattening

Convert nested subqueries to joins when possible.

Original:

```sql
SELECT *
FROM Orders
WHERE customer_id IN (SELECT customer_id FROM Customers WHERE region = 'North');
```

Transformed:

```sql
SELECT O.*
FROM Orders O JOIN Customers C ON O.customer_id = C.customer_id
WHERE C.region = 'North';
```

2.3 Estimating Expected Size

Accurate size estimation is crucial for choosing efficient execution plans. The query optimizer uses statistics and heuristics to estimate:

- The number of tuples in intermediate results
- The size of intermediate results in bytes
- The distribution of values

2.3.1 Selectivity Factors

Selectivity represents the fraction of tuples that satisfy a predicate.

For equality predicates (A = constant):

- If statistics are available: selectivity = 1 / number of distinct values

- Default assumption: selectivity = 1/10 (10% of tuples match)

For range predicates (A > constant):

- Estimate based on the value distribution
- For uniform distribution: selectivity = (max value - constant) / (max value - min value)

For conjunctive predicates (A AND B):

- Selectivity(A AND B) = Selectivity(A) × Selectivity(B)
  - This assumes independence, which may not always be accurate

For disjunctive predicates (A OR B):

- Selectivity(A OR B) = Selectivity(A) + Selectivity(B) - Selectivity(A AND B)

2.3.2 Join Size Estimation

For natural joins R ⋈ S:

- Size estimation = |R| × |S| × join selectivity
- Join selectivity typically calculated as: 1 / max(distinct values in R, distinct values in S)

Example: For a join between Orders (1000 tuples) and Customers (100 tuples) on customer_id:

- If Customers has 100 distinct customer_id values
- Expected join size = 1000 × 100 × (1/100) = 1000 tuples

2.4 Using Statistics in Estimation

Database systems collect and maintain statistics to improve estimation accuracy.

2.4.1 Types of Statistics

1. Table Statistics

- Table cardinality (number of tuples)
- Table size in blocks
- Average tuple size

2. Column Statistics

- Number of distinct values
- Min/max values
- Null value count
- Data distribution (histograms)

3. Index Statistics

- Index height
- Index cardinality
- Clustering factor

2.4.2 Histograms

Histograms represent the distribution of values in a column, enabling more accurate selectivity estimation.

Types of histograms:

Equi-width histograms

- Divide the value range into equal-width buckets
- Easy to create and maintain
- May not handle skewed data well

Equi-depth (equi-height) histograms

- Each bucket contains approximately the same number of tuples
- Better for skewed distributions
- More complex to maintain

Example: For ages ranging from 18 to 70:

Equi-width histogram (5 buckets):

```
Bucket 1 (18-28): 500 tuples
Bucket 2 (29-39): 1200 tuples
Bucket 3 (40-50): 800 tuples
Bucket 4 (51-61): 300 tuples
Bucket 5 (62-72): 200 tuples
```

2.4.3 Statistics Collection

Database systems collect statistics through:

1. Automatic Collection

- During database maintenance windows
- After significant changes to the data

2. Manual Collection

```
-- Oracle
ANALYZE TABLE orders COMPUTE STATISTICS;

-- PostgreSQL
ANALYZE orders;

-- SQL Server
UPDATE STATISTICS orders;
```

3. Sampling Methods

- Random sampling
- Stratified sampling
- Dynamic sampling during query execution

2.5 Query Improvement Strategies

Query improvement aims to enhance query performance through various techniques. This includes selecting appropriate indexes, optimizing join methods, using materialized views, and leveraging parallel query execution.

2.5.1 Index Selection

Choosing appropriate indexes can dramatically improve query performance:

Types of indexes:

- B-tree indexes (general-purpose)
- Hash indexes (equality predicates)
- Bitmap indexes (low-cardinality columns)
- R-tree indexes (spatial data)

Considerations for index selection:

- Query patterns (which columns are frequently filtered or joined)
- Update frequency (indexes slow down updates)
- Index selectivity
- Storage requirements

Example:

```
-- Creating an index for frequently queried columns
CREATE INDEX idx_customer_region_city ON Customers(region, city);
```

2.5.2 Join Methods

Different join algorithms are suited for different scenarios:

Nested Loop Join

- Good for small tables or indexed joins
- For each tuple in the outer table, scan the inner table
- Time complexity: O(n × m) without indexes

Hash Join

- Effective for large tables without useful indexes
- Build a hash table on the smaller table, probe with the larger table
- Time complexity: O(n + m)

Sort-Merge Join

- Good when inputs are already sorted or sorting is needed anyway
- Sort both relations on join attributes, then merge
- Time complexity: O(n log n + m log m)

2.5.3 Materialized Views

Materialized views store precomputed query results to improve performance for complex queries:

```sql
CREATE MATERIALIZED VIEW monthly_sales AS
SELECT product_id, EXTRACT(MONTH FROM sale_date) as month,
       SUM(quantity) as total_quantity, SUM(amount) as total_amount
FROM Sales
GROUP BY product_id, EXTRACT(MONTH FROM sale_date);
```

Benefits:

- Faster query performance for complex aggregations and joins
- Reduced computational load during peak hours

Challenges:

- Need to be refreshed when source data changes
- Require additional storage

2.5.4 Parallel Query Execution

Dividing query execution across multiple processors or nodes:

Intra-query parallelism:

- Executing different parts of the same query in parallel

- Includes parallel joins, scans, and aggregations

Inter-query parallelism:

- Executing multiple queries simultaneously
- Important for high-concurrency OLTP systems

2.6 Query Evaluation

Query evaluation is the process of executing the optimized query plan by the database system to retrieve results. It involves accessing data, applying operations, and returning results to the user.

2.6.1 Physical Operators

Database systems implement relational algebra operations as physical operators:

Table Scan Operators

- Full table scan
- Index scan
- Index-only scan
- Sample scan

Join Operators

- Nested loops join
- Hash join
- Merge join

Aggregation Operators

- Sort-based aggregation
- Hash-based aggregation

Sort Operators

- External sort
- In-memory sort

2.6.2 Iterator Model

Most database systems use the iterator model (or Volcano model) for query execution:

- Each operator implements three methods:
    - open(): Initialize the operator
    - next(): Produce the next tuple

    - close(): Release resources

- Operators are combined to form a pipeline
- Each operator pulls data from its child operators

```
+-----------------+
| SELECT operator |
+-----------------+
        ↑
        |
+-----------------+
|  JOIN operator  |
+-----------------+
    ↑         ↑
    |         |
+--------+  +--------+
| Scan A|  | Scan B|
+--------+  +--------+
```

2.7 View Processing

Views are virtual tables defined by a query. They provide a way to simplify complex queries, encapsulate logic, and enhance security by restricting access to specific data. Views can be used in queries just like regular tables, but they do not store data themselves. Instead, they store the SQL query that defines them.

2.7.1 View Definition and Types

Creating views:

```
CREATE VIEW HighValueCustomers AS
SELECT customer_id, name, email, SUM(order_amount) as total_spent
FROM Customers JOIN Orders USING (customer_id)
GROUP BY customer_id, name, email
HAVING SUM(order_amount) > 10000;
```

Types of views:

- **Regular views**: Not physically materialized
- **Materialized views**: Physically stored for better performance

2.7.2 View Expansion

When a query references a view, the DBMS can use two approaches:

1. View expansion (substitution)

- Replace the view reference with its definition
- Optimize the expanded query as a whole

Example: Query:

```sql
SELECT name, total_spent
FROM HighValueCustomers
WHERE total_spent > 20000;
```

Expanded:

```sql
SELECT name, SUM(order_amount) as total_spent
FROM Customers JOIN Orders USING (customer_id)
GROUP BY customer_id, name, email
HAVING SUM(order_amount) > 10000 AND SUM(order_amount) > 20000;
```

2. View materialization

- Compute the view result first
- Apply the outer query conditions to the result

2.8 Query Processor

The query processor is the DBMS component responsible for translating, optimizing, and executing queries. It plays a crucial role in determining how efficiently a query can be executed. The query processor consists of several components, each responsible for different aspects of query processing. The main components include the parser, optimizer, and executor.

2.8.1 Query Processor Architecture

```
┌──────────────────────────────────────────────────┐
│                Query Processor                   │
├──────────────────────────────────────────────────┤
│ ┌─────────────┐ ┌──────────────┐ ┌────────────┐  │
│ │   Parser    │ │  Optimizer   │ │  Executor  │  │
│ │ - Scanning  │ │ - Algebraic  │ │ - Operators│  │
│ │ - Parsing   │ │   Rewriting  │ │ - Buffer   │  │
│ │ - Semantic  │ │ - Plan       │ │   Manager  │  │
│ │   Checking  │ │   Generation │ │ - I/O      │  │
│ └─────────────┘ └──────────────┘ └────────────┘  │
└──────────────────────────────────────────────────┘
```

Components of the query processor:

1. **Parser**

    - Lexical analysis (tokenizing)
    - Syntax analysis (parsing)
    - Semantic analysis
    - Internal representation creation

2. **Optimizer**

    - Logical optimization (query rewriting)
    - Physical optimization (plan selection)
    - Cost estimation

3. **Executor**

    - Plan interpretation
    - Operator execution
    - Result delivery

2.8.2 Query Processor Functions

The query processor performs several key functions:

- **Parsing**: Converts SQL queries into an internal representation (parse tree or abstract syntax tree).
- **Optimization**: Analyzes the query to find the most efficient execution plan.
- **Execution**: Executes the chosen plan using physical operators.
- **Result Delivery**: Returns the final result to the user or application.
