

**Course Material**  
**of**  
**Database Management Systems**

Subject Code: CS 313  
Class: ICD Vth Semester

**Prepared by:**

Dr. Jagdeep Singh

Assistant Professor (CSE)



**Department**  
**of**  
**Computer Science and Engineering**

**Sant Longowal Institute of Engineering and Technology, Longowal, India**

October 2024

## Syllabus

Title of the course	: <b>Database Management Systems</b>	
Subject Code	: <b>CS-313</b>	
Weekly load	: 7 Hrs	LTP 3-0-4
Credit	: 5 (Lecture 3; Practical 2)	

**Course Outcomes:** At the end of the course, the student will be able to:

CO1	Understand functional components of the DBMS.
CO2	Design database schema and study different data models.
CO3	Understand the concept of normalization.
CO4	Understand the concepts of PL/SQL.

CO/PO Mapping : (Strong(S)/Medium(M)/Weak(W) indicates strength of correlation)										
COs	Programme Outcomes (POs)									
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10
CO1		S								S
CO2		S								S
CO3		S								S
CO4		S	M	M						S

### Theory

Unit	Main Topics	Course outlines	Lecture(s)
<b>Unit-1</b>	1. Introduction	Database Systems, Database and its purpose, Characteristics of the database approach, Advantages and disadvantages of database systems.	06
	2. Classification of DBMS Users	Classification of DBMS Users; Actors on the scene, Database Administrators, Database Designers, End Users, System Analysts and Application Programmers, Workers behind the scene	06
	3. Database System Concepts and Architecture	Data models, schemas, instances, data base state. DBMS Architecture; The External level, The conceptual level, The internal level	06
	4. Mappings	Mappings. Data Independence; Logical data Independence, Physical data Independence.	03
<b>Unit-2</b>	5. Data Models	Relational Data Model, Network Data Model, Hierarchical	08

		Model	
	6. Data Modeling using E.R. Model	Entities and Attributes, Entity types and Entity sets, attribute and domain of attributes, Relationship among entities.	05
	7. Keys	Key, Different types of keys, Integrity Principles.	06
	8. Normalization	Functional dependencies, First, Second and Third normal forms, Boyce/Codd normal form.	08
	Advanced Topics	Big Data, Data Analytics, Physical Storage Systems, Data Storage Structures, Indexing, Transactions, Recovery Systems, PL/SQL	

**Total=48**

**TEXT BOOKS:**

1. Data base System Concepts, Silberschatz, Korth, McGraw hill, Sixth Edition.
2. Data base Management Systems, Raghurama Krishnan, Johannes Gehrke, TATA McGraw Hill 3rd Edition.

**REFERENCE BOOKS:**

1. Fundamentals of Database Systems, Elmasri Navathe Pearson Education.
2. An Introduction to Database systems, C.J. Date, A.Kannan, S.Swami Nadhan, Pearson, Eight Edition

## Unit: 1: Introduction to Database Systems

---

A Database Management System (DBMS) is a sophisticated software system that allows users to store, manage, and retrieve large amounts of data efficiently. DBMSs are crucial in organizing data for enterprises of all sizes, providing a structure for data storage and access.

Various industries rely on database applications to manage their day-to-day operations:

- Banking: Tracks every customer transaction.
- Airlines: Manages flight reservations and schedules.
- Universities: Records student registrations and grades.
- Sales: Tracks customer orders and generates personalized recommendations.
- Manufacturing: Manages production, inventory, orders, and the supply chain.
- Human Resources: Stores employee records, including salaries and tax deductions.

In today's data-driven world, databases are essential to managing almost every aspect of life, from online shopping to finance management.

### **Purpose of Database Systems**

Before the advent of DBMSs, applications were built on top of traditional file systems. However, this posed several significant challenges:

- Data Redundancy and Inconsistency: Different files stored the same data in multiple formats, leading to discrepancies.
- Difficulty Accessing Data: Every new task required a new program to retrieve data.
- Data Isolation: Related data was stored in different files and formats.
- Integrity Problems: Business rules were embedded in program code and hard to enforce.
- Atomicity Issues: Failures could result in partial updates, leaving data inconsistent.
- Concurrent Access: Multiple users accessing the same data led to conflicts.
- Security Issues: It was difficult to restrict access to sensitive information.

Database systems address these issues by offering structured data management, ensuring consistency, security, and efficient access.

### **Levels of Abstraction**

A DBMS uses multiple levels of abstraction to manage data efficiently. These include:

- Physical Level: Describes how data is physically stored on hardware.
- Logical Level: Describes the structure of the data and relationships between data types.
- View Level: Provides different perspectives of the data, hiding complex details.

At the physical level, the DBMS defines how the data is stored on disk, while the logical level focuses on the organization of data into tables and relationships. The view level simplifies data for users by providing customized views of the data, such as only showing relevant fields to specific users.

### Instances and Schemas

Schemas define the logical structure of the database, acting as blueprints for organizing data. They specify what data is stored and how it is related but do not include the actual data.

- Physical Schema: Defines how the data is stored physically.
- Logical Schema: Defines how data is logically organized, such as tables and relationships.

An instance refers to the actual data stored in the database at a given moment, while the schema remains constant. Physical Data Independence allows changes to the physical schema without affecting the logical schema.

### Data Models

A data model is a collection of tools used to describe the structure, relationships, semantics, and constraints of data. Common types of data models include:

- Relational Model: Organizes data into tables (relations) of rows and columns.
- Entity-Relationship (ER) Model: Uses entities and relationships to represent data.
- Object-Based Models: Incorporate object-oriented concepts into databases.
- XML: Used to represent hierarchical and semi-structured data.

### Data Manipulation Language (DML)

DML is a language used to access and manipulate data stored in a database. It consists of two main types:

- Procedural DML: The user specifies both what data is needed and how to get it.
- Non-Procedural DML: The user specifies only what data is needed, while the DBMS decides how to retrieve it.

SQL (Structured Query Language) is the most widely used non-procedural DML. For example, to find the name of a customer with a specific ID, you would write the following query:

```
SELECT customer_name FROM customer WHERE customer_id = '001';
```

### SQL

SQL is the standard language for interacting with relational databases. It allows users to retrieve and manipulate data using simple queries.

For example, to find the balance of accounts held by a customer with ID '001':

```
SELECT account.balance FROM account JOIN depositor ON depositor.account_number =  
account.account_number WHERE depositor.customer_id = '001';
```

## Introduction to the Relational Model

---

### Structure of Relational Databases

In a relational database, data is stored in tables called relations. A relation consists of rows (tuples) and columns (attributes), where each row represents a record and each column represents a property of the record. For example, consider the following table representing instructors at a university.

ID	Name	Department	Salary
101	Alice Johnson	Physics	85,000
102	Bob Smith	Computer Science	92,000
103	Carol White	Mathematics	78,000

In this relation, each row represents an instructor, and the columns represent the attributes of the instructors such as their ID, name, department, and salary.

### Database Schema

A database schema defines the logical structure of the database. It specifies the tables and the attributes that make up those tables. For example, the schema for the 'instructor' table could be described as:

```
instructor(ID, name, dept_name, salary)
```

A database instance refers to the actual data stored in the database at a given point in time. While the schema remains constant, the instance can change as data is added, updated, or deleted.

### Keys

In a relational database, keys are crucial for identifying unique records in a relation. There are several types of keys:

**Superkey:** A set of attributes that can uniquely identify a tuple. For example, {ID} is a superkey for the 'instructor' relation.

**Candidate key:** A minimal superkey. {ID} is a candidate key because it is the minimal set of attributes needed to uniquely identify an instructor.

**Primary key:** A candidate key chosen to be the main key for the relation. In this case, {ID} is the primary key.

**Foreign key:** An attribute in one relation that refers to the primary key of another relation. For example, the 'dept\_name' in the 'instructor' table could reference the 'department' table.

### Relational Algebra

Relational algebra is a procedural query language that allows the manipulation of data in relational databases. It consists of a set of operations that take one or more relations as input and produce a new relation as output. The basic operations in relational algebra include selection, projection, union, set difference, Cartesian product, and renaming.

**Select Operation ( $\sigma$ )**

The select operation retrieves tuples that satisfy a given condition. For example, to find all instructors who work in the 'Physics' department, the following query can be used:

$$\sigma \text{ dept\_name} = \text{'Physics'} (\text{instructor})$$

This operation retrieves all tuples where the 'dept\_name' is 'Physics'.

**Project Operation ( $\Pi$ )**

The project operation retrieves specified columns from a relation. For example, to find the ID and salary of all instructors, we use the following query:

$$\Pi \text{ ID, salary} (\text{instructor})$$

This operation will return a relation containing only the 'ID' and 'salary' attributes of the instructor table.

**Cartesian Product ( $\times$ )**

The Cartesian product combines tuples from two relations. For instance, if we have 'instructor' and 'teaches' relations, the Cartesian product combines each tuple from the instructor table with every tuple from the teaches table.

Example of Cartesian product between 'instructor' and 'teaches' relations:

Instructor ID	Instructor Name	Course ID	Year
101	Alice Johnson	CS101	2022
102	Bob Smith	MATH203	2022

**Join Operation ( $\bowtie$ )**

The join operation is used to combine two relations based on a common attribute. For example, to join the 'instructor' and 'teaches' relations based on the instructor's ID, we write the following query:

$$\text{instructor} \bowtie \text{instructor.ID} = \text{teaches.ID} \text{ teaches}$$

This operation combines the rows from 'instructor' and 'teaches' where the instructor's ID matches.

**Union, Set Difference, and Set Intersection**

The union operation combines two relations into one. Set difference finds the tuples that exist in one relation but not in the other, while set intersection finds tuples that exist in both relations.

Example query for union operation:

To find all courses taught in either Fall 2021 or Spring 2022, we can write:

$$\Pi \text{ course\_id} (\sigma \text{ semester} = \text{'Fall'} \text{ AND } \text{year} = 2021 (\text{section})) \cup \Pi \text{ course\_id} (\sigma \text{ semester} = \text{'Spring'} \text{ AND } \text{year} = 2022 (\text{section}))$$

## SQL: Structured Query Language

---

### History of SQL

SQL, initially known as SEQUEL, was developed as part of IBM's System R project at the San Jose Research Laboratory. It was later renamed to Structured Query Language (SQL) and standardized by ANSI and ISO. Over time, several major versions of SQL were introduced:

- **SQL-86**
- **SQL-89**
- **SQL-92**
- **SQL:1999** (made Y2K-compliant)
- **SQL:2003**

Most modern commercial database systems support features from SQL-92, along with custom features specific to each system.

### SQL Components

SQL is divided into several core components, each handling specific database operations:

- **Data Manipulation Language (DML):** Allows querying and modifying data (e.g., inserting, updating, deleting records).
- **Data Definition Language (DDL):** Commands for defining schemas, setting up tables, and specifying integrity constraints.
- **View Definition:** Commands for creating views, which are virtual tables derived from queries.
- **Transaction Control:** Manages the start and end of transactions, ensuring consistency.
- **Embedded SQL:** Enables embedding SQL statements within programming languages.
- **Authorization:** Specifies access rights for tables and views, allowing or restricting user operations.

### Data Definition Language (DDL)

DDL allows users to define and modify the database schema, including:

- **Schema Definitions:** Structure for each table.
- **Attribute Data Types:** Specifies the type of data (e.g., integer, varchar).
- **Integrity Constraints:** Ensures consistency (e.g., primary and foreign keys).
- **Indexing:** Specifies which fields should be indexed for efficient access.
- **Authorization:** Controls user access and permissions.
- **Physical Storage:** Defines how data is stored on disk.

### Domain Types in SQL

SQL supports a variety of data types to represent different kinds of data, such as:



- **char(n)**: Fixed-length character strings.
- **varchar(n)**: Variable-length character strings.
- **int**: Integer values.
- **smallint**: A smaller subset of integer values.
- **numeric(p, d)**: Fixed-point numbers with p digits and d digits after the decimal (e.g., numeric(3,1)).
- **real** and **double precision**: Floating-point numbers.
- **float(n)**: Floating-point numbers with at least n digits of precision.

## Creating Tables in SQL

Tables in SQL are created with the create table command. Here's a general syntax example:

```
sql
```

```
create table table_name (  
    column1 data_type,  
    column2 data_type,  
    ... ,  
    primary key (column_name),  
    foreign key (column_name) references other_table(column_name)  
);
```

**Example:** Creating an instructor table:

```
sql
```

```
create table instructor (  
    ID char(5),  
    name varchar(20),  
    dept_name varchar(20),  
    salary numeric(8,2),  
    primary key (ID)  
);
```

## Integrity Constraints

SQL provides integrity constraints to maintain data accuracy and consistency:

- **Primary Key**: Ensures each record has a unique identifier.
- **Foreign Key**: Enforces a relationship between two tables by referencing the primary key of another table.
- **Not Null**: Prevents fields from having null (unknown) values.

## Basic Query Structure

A typical SQL query follows this structure:

sql

```
select column1, column2  
from table1, table2  
where condition;
```

**Example:** Retrieving all instructor names from an instructor table:

sql

```
select name  
from instructor;
```

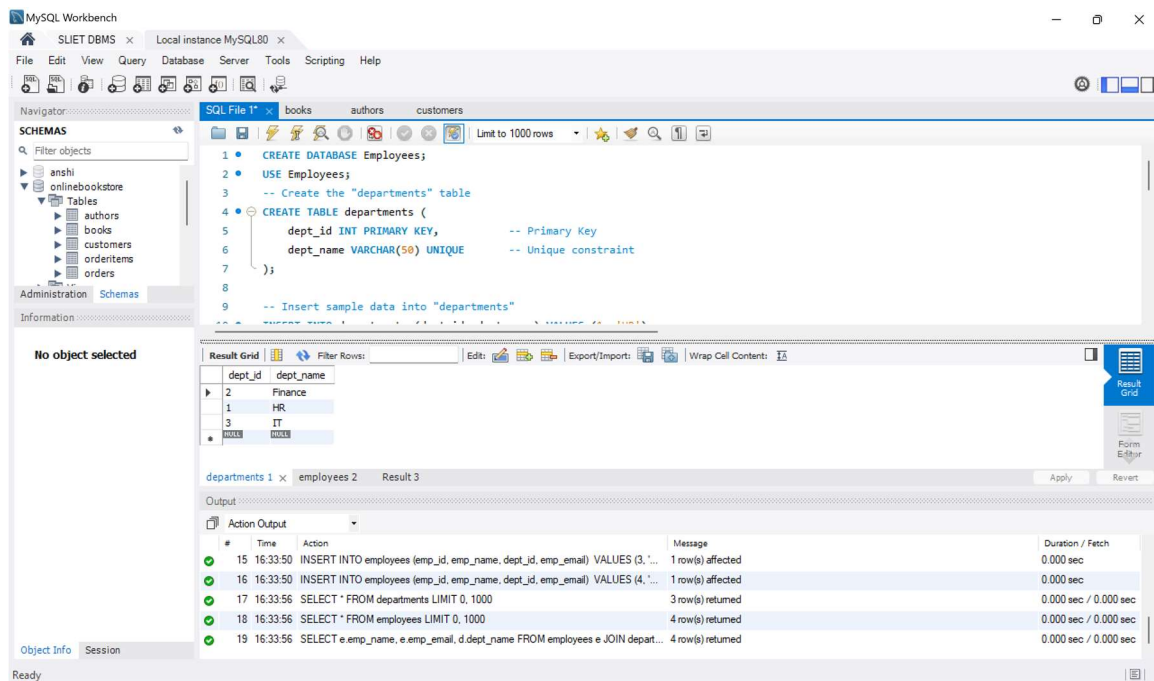


Fig.1 : Create database

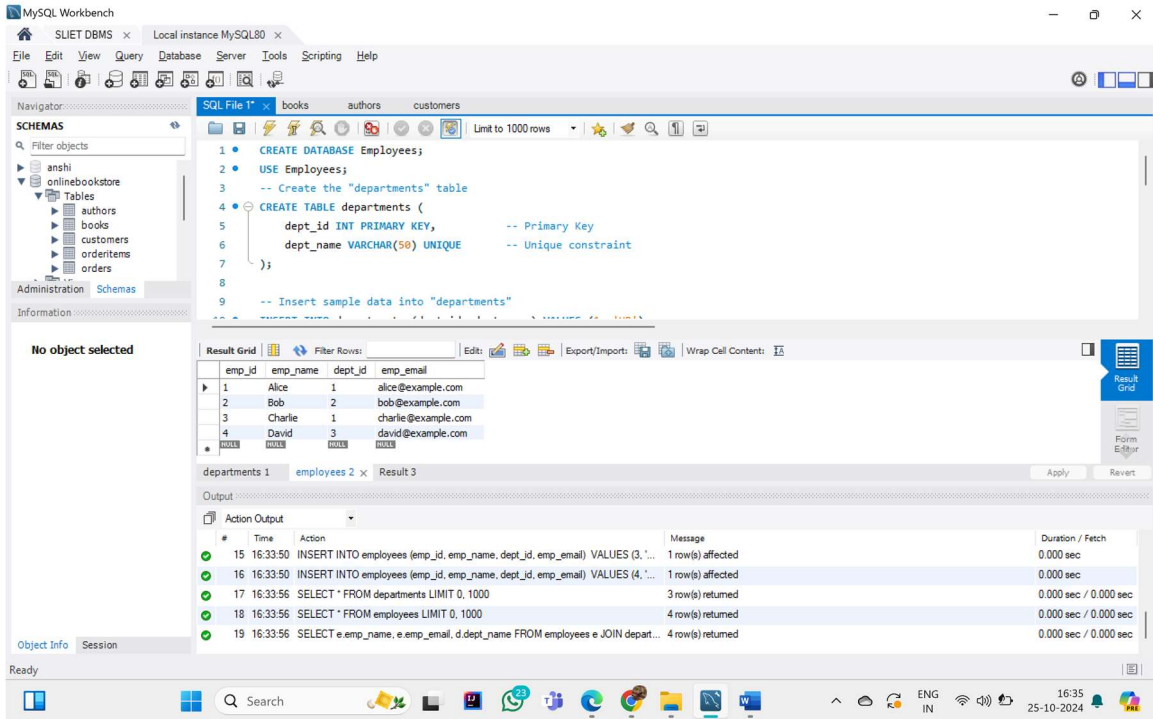


Fig. 2: Create Table

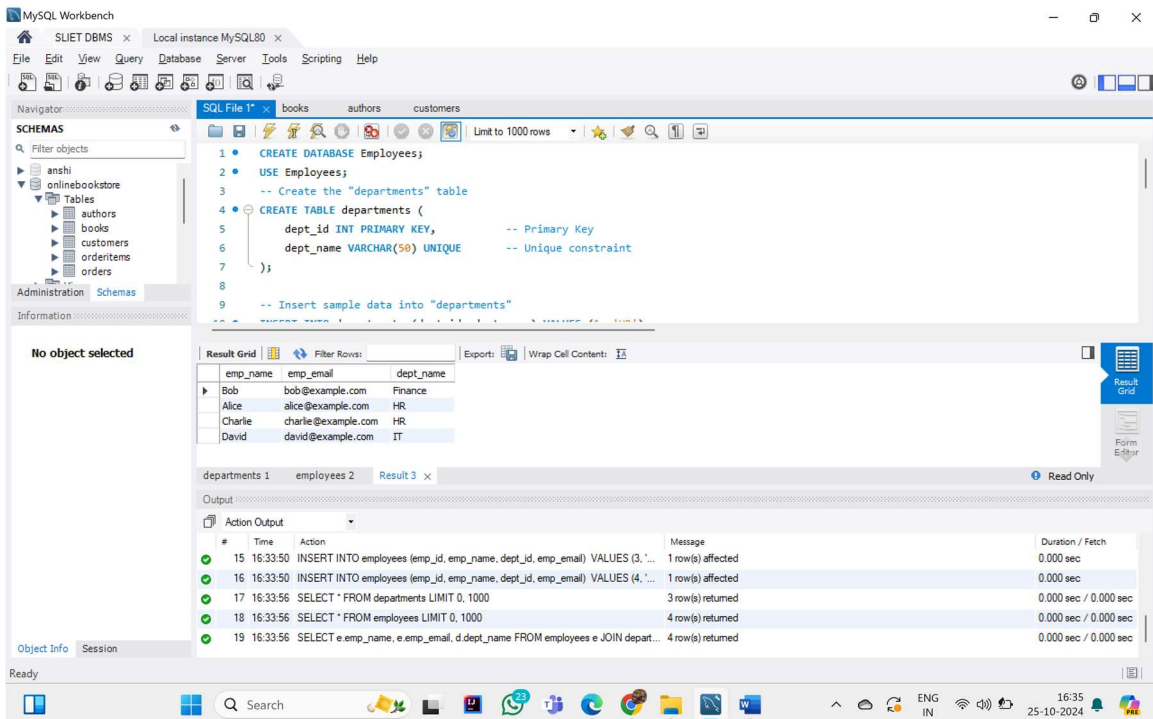


Fig. 3. Insert into command

### The select Clause

The select clause specifies which columns to retrieve in the result set.

**Example:** Listing department names from the instructor table, without duplicates:

```
sql  
  
select distinct dept_name  
from instructor;
```

SQL is **case-insensitive**, meaning Name, NAME, and name are all equivalent.

---

### Arithmetic Expressions in the select Clause

Arithmetic operations can be performed within the select clause.

**Example:** Calculating monthly salary by dividing the salary field by 12:

```
sql  
  
select ID, name, salary / 12 as monthly_salary  
from instructor;
```

---

### The where Clause

The where clause filters records based on specific conditions.

**Example:** Finding all instructors in the "Computer Science" department:

```
sql  
  
select name  
from instructor  
where dept_name = 'Computer Science';
```

---

### The from Clause and Cartesian Product

The from clause lists tables involved in the query and generates a Cartesian product if multiple tables are included.

**Example:** Combining data from instructor and teaches tables:

```
sql  
  
select *  
from instructor, teaches;
```

## Join Operations

Joins in SQL combine records from two or more tables based on related columns.

**Example:** Listing names of instructors who taught specific courses:

```
sql

select name, course_id
from instructor join teaches on instructor.ID = teaches.ID;
```

## Set Operations

SQL allows combining the results of multiple queries using set operations:

- **union:** Combines results from two queries, eliminating duplicates.
- **intersect:** Finds common records between two queries.
- **except:** Returns records present in one query but not the other.

**Example:** Finding courses offered in both Fall 2017 and Spring 2018:

```
sql

(select course_id from section where semester = 'Fall' and year = 2017)
intersect
(select course_id from section where semester = 'Spring' and year = 2018);
```

## Null Values

SQL supports null values to represent missing or unknown data. Arithmetic operations with null yield null results.

**Example:** Finding instructors with unknown salary:

```
sql

select name
from instructor
where salary is null;
```

## Aggregate Functions

SQL includes several aggregate functions:

- **avg:** Calculates the average value.
- **min** and **max:** Find the minimum and maximum values, respectively.
- **sum:** Adds up all values in a column.
- **count:** Counts the number of records.

**Example:** Calculating the average salary in the "Computer Science" department:

sql

```
select avg(salary)
from instructor
where dept_name = 'Computer Science';
```

### Grouping and the Having Clause

Grouping allows applying aggregate functions to subsets of data, while the having clause filters groups based on aggregate conditions.

**Example:** Finding departments where the average salary exceeds \$42,000:

sql

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name
having avg(salary) > 42000;
```

### Nested Queries and Subqueries

SQL supports subqueries, which are queries nested within another query. They allow complex filtering and calculations within a single query structure.

**Example:** Listing instructors who earn more than the average salary of all instructors:

sql

```
select name
from instructor
where salary > (select avg(salary) from instructor);
```

---

### Modifications in SQL

SQL allows data manipulation via commands for inserting, updating, and deleting records.

#### 1. Insertion:

sql

```
insert into instructor values ('10211', 'Smith', 'Biology', 66000);
```

#### 2. Deletion:

sql

```
delete from instructor where dept_name = 'Finance';
```

### 3. Updating:

```
sql
```

```
update instructor set salary = salary * 1.05 where salary < 70000;
```

## Intermediate SQL

This class introduces advanced SQL concepts such as join operations, views, transactions, integrity constraints, data types, indexing, and authorization mechanisms. These concepts are key for performing more sophisticated database queries and operations.

### Join Operations

In SQL, **join operations** are used to combine data from two or more tables based on related columns. The result is a new table containing data that meets the join condition. Join operations include:

1. **Natural Join:** Matches rows with the same values in columns that share the same name in both tables. For example, if we want to list students and the courses they are enrolled in:

```
select name, course_id  
from student natural join takes;
```

2. **Inner Join:** Retrieves rows that have matching values in both tables using the on keyword to specify the condition. To find students and the titles of the courses they are enrolled in:

```
select name, title  
from student inner join course on takes.course_id = course.course_id;
```

3. **Outer Join:** Ensures that all rows from one or both tables are returned, even if there are no matching rows in the other table. Types include:
  - **Left Outer Join:** Returns all rows from the left table, with null for missing matches from the right table.
  - **Right Outer Join:** Returns all rows from the right table, with null for missing matches from the left table.
  - **Full Outer Join:** Returns rows from both tables, including unmatched rows with null values.

Example of a left outer join:

```
select course_id, title  
from course left outer join prereq on course.course_id = prereq.course_id;
```

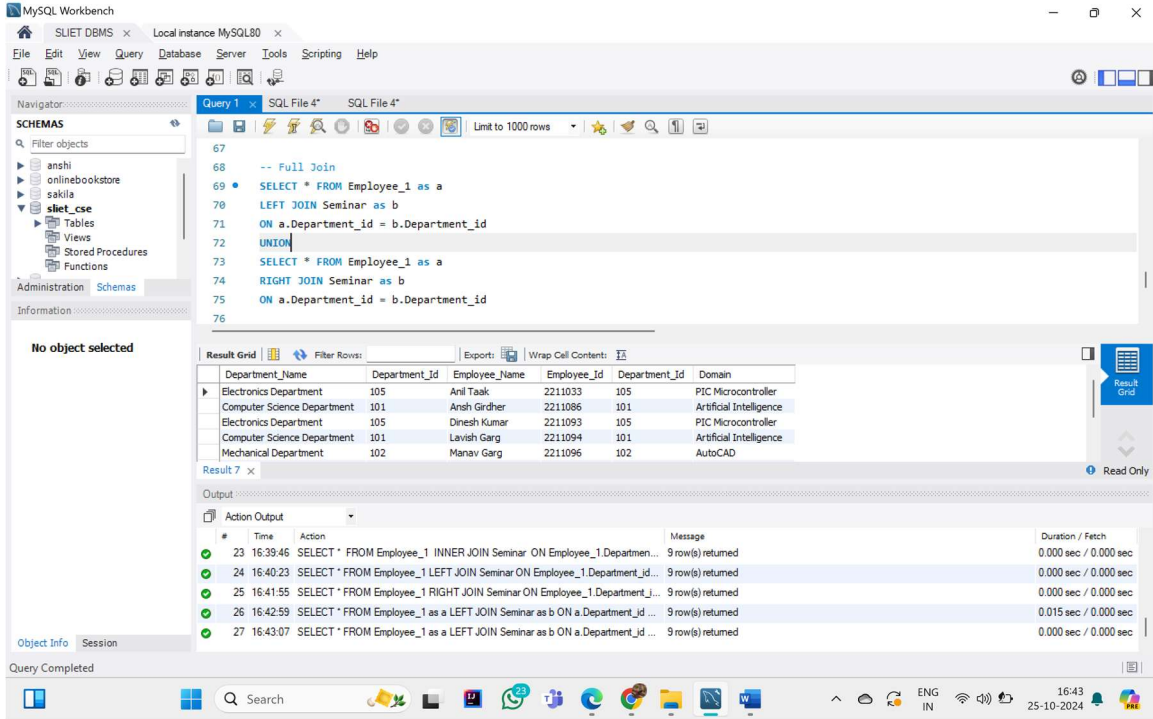


Fig. 4: Full Join

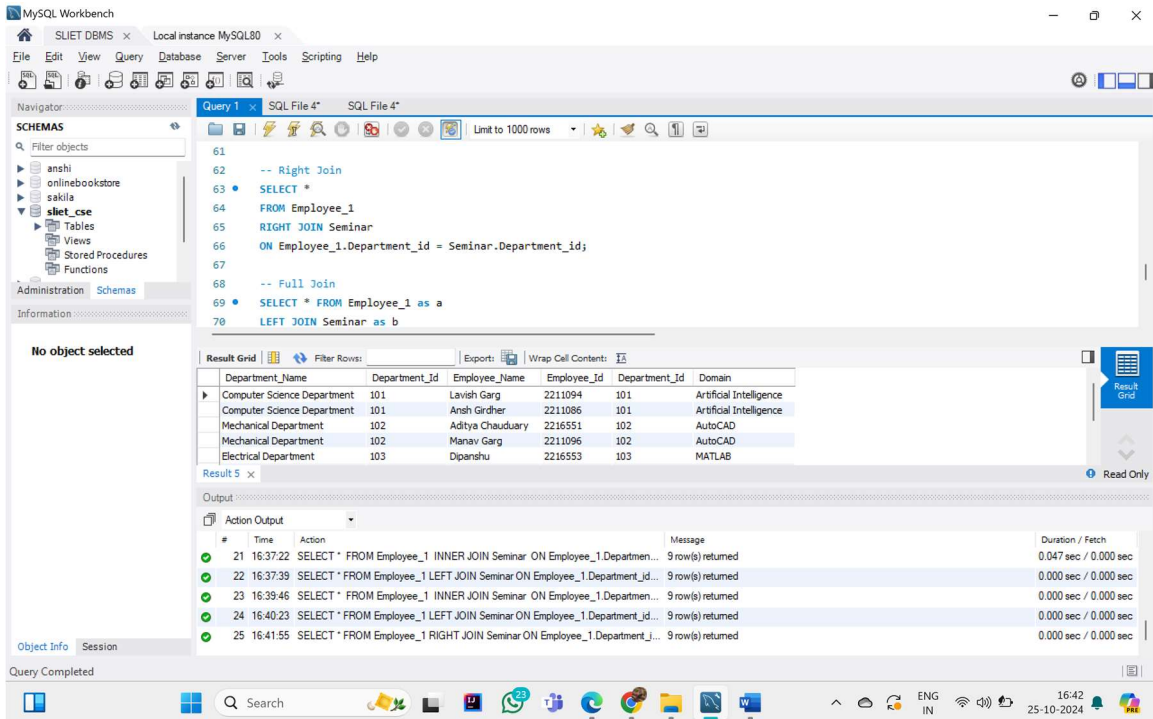


Fig.5: Outer Join



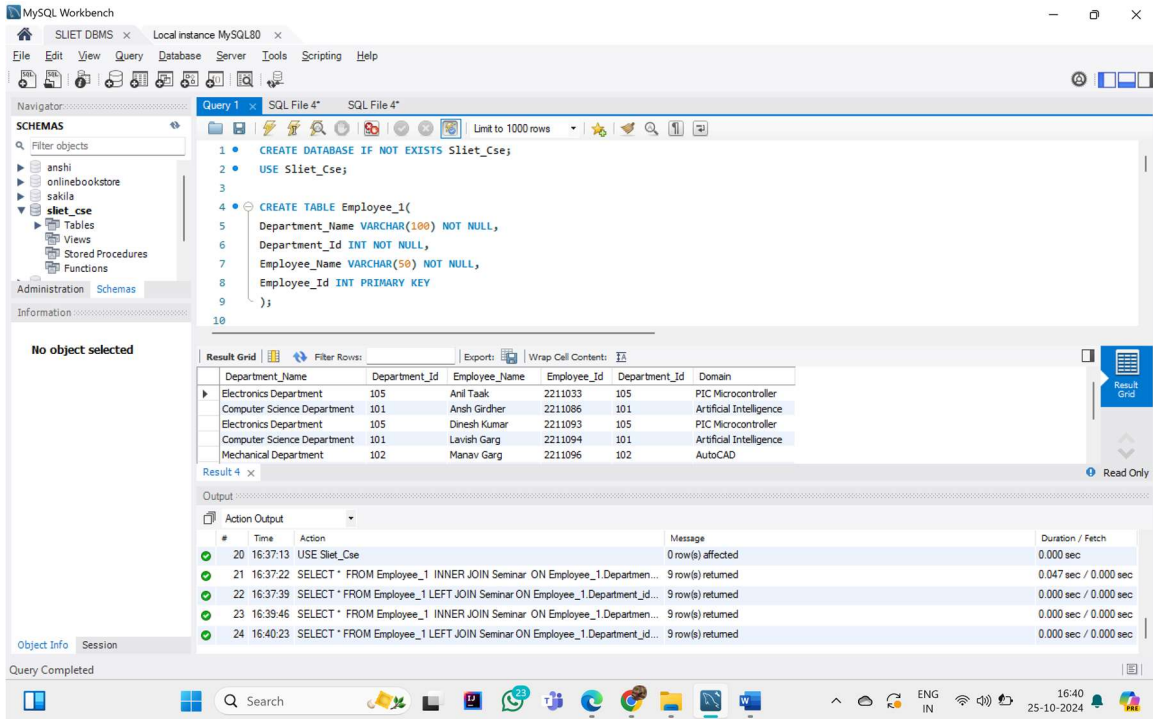


Fig.6: Left Join

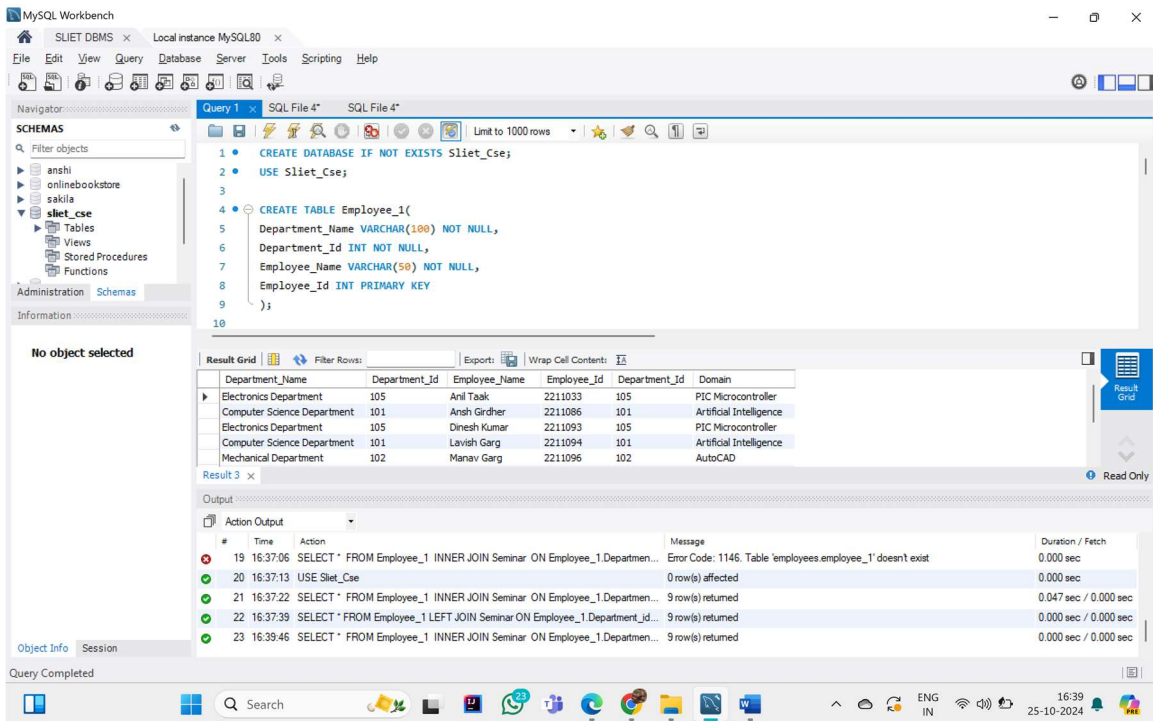


Fig. 7: Inner Join

## Views in SQL

A **view** is a virtual table based on the result of a query. Views allow users to see data without revealing the full database structure. For example, to create a view that lists instructors without their salaries:

```
create view faculty as
select ID, name, dept_name
from instructor;
```

To retrieve instructors in the Biology department:

```
select name
from faculty
where dept_name = 'Biology';
```

You can also create a view based on another view, such as restricting instructors to those in a specific building:

```
create view bio_instructors_watson as
select name
from faculty
where building = 'Watson';
```

---

## Transactions in SQL

A **transaction** in SQL is a sequence of statements that are treated as a single unit of work, ensuring database consistency. Transactions follow the ACID properties and can either be committed (making changes permanent) or rolled back (undoing changes).

Example transaction:

```
begin transaction;
update account set balance = balance - 100 where account_id = 'A001';
update account set balance = balance + 100 where account_id = 'A002';
commit;
```

---

## Integrity Constraints

**Integrity constraints** ensure the consistency and accuracy of data. Common integrity constraints include:

1. **Primary Key:** Ensures uniqueness for each record in a table.
2. **Foreign Key:** Ensures that values in one table correspond to valid values in another table.
3. **Check Constraint:** Specifies a condition that must hold true for all records.

For instance, here's a check constraint that ensures semester values are valid:

```
create table section (
course_id varchar(8),
```

```
semester varchar(6),  
check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))  
);
```

---

### Referential Integrity

Referential integrity ensures that relationships between tables are consistent. When a foreign key is defined in one table, it must correspond to a valid primary key in another table.

If a department is referenced in the instructor table, deleting that department might cause inconsistency unless we use cascading actions. For example:

```
create table course (  
dept_name varchar(20),  
foreign key (dept_name) references department  
on delete cascade  
on update cascade  
);
```

---

### Indexing in SQL

**Indexes** are used to speed up query processing by enabling quicker access to rows. An index acts like a lookup table that the database uses to find data more efficiently.

For instance, creating an index on the ID column of the student table:

```
create index studentID_index on student(ID);
```

With this index, a query searching for a student by ID will be faster:

```
select *  
from student  
where ID = '12345';
```

---

### Authorization in SQL

SQL provides authorization mechanisms to control what users can do with data. Types of authorizations include:

- **Read:** Allows users to read data.
- **Insert:** Allows users to add data.
- **Update:** Allows users to modify data.
- **Delete:** Allows users to remove data.

To grant select permission on the department table to user Amit, we can write:

```
grant select on department to Amit;
```

---

## Roles in SQL

**Roles** allow administrators to group permissions together and assign them to users collectively. For instance, we can create a role called instructor and assign permissions to it:

```
create role instructor;  
grant select on takes to instructor;  
grant instructor to Amit;
```

Amit now inherits all the permissions assigned to the instructor role.

## Advanced SQL

This class covers advanced SQL features, such as integrating SQL with programming languages, creating functions and triggers, handling recursive queries, working with advanced aggregation, and using Online Analytical Processing (OLAP). These tools enable SQL users to manage complex data scenarios and execute sophisticated queries.

---

## Accessing SQL from a Programming Language

SQL is a declarative language designed for querying databases, but it doesn't offer the same control flow and flexibility as a general-purpose programming language. To combine SQL with more dynamic logic, it's often integrated into other programming languages such as Java, Python, or C++. Two common ways to achieve this integration are:

- **Embedded SQL:** SQL statements are directly embedded in the host language's code. When compiled, these SQL statements are converted into function calls to the database.
- **API-based SQL Access:** APIs (like JDBC for Java) allow the program to send SQL queries to the database at runtime and process the results.

### *Java Database Connectivity (JDBC)*

In Java, JDBC is the API that facilitates interaction with databases. It allows you to send SQL commands and process the returned data. Here's an example of how to establish a connection and execute a query using JDBC:

```
java  
  
public static void JDBCexample(String dbid, String userid, String passwd) {  
    try (Connection conn = DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/school", userid, passwd);  
        Statement stmt = conn.createStatement();) {  
        // Execute SQL queries and handle results  
    } catch (SQLException sqle) {
```

```

        System.out.println("SQLException: " + sqle);
    }
}

```

In Python, a similar connection to the database can be achieved using the sqlite3 library:

```

python

import sqlite3

def get_student_data(db_name):
    conn = sqlite3.connect(db_name)
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM students WHERE gpa > 3.5")
    for row in cursor.fetchall():
        print(row)
    conn.close()

```

This Python example retrieves data from the students table where the GPA is greater than 3.5.

## Functions and Procedures in SQL

SQL allows you to create **functions** and **procedures** that can store reusable logic directly within the database. Functions return a value, while procedures may perform actions but do not return a value.

### *SQL Function Example*

Consider a scenario where you need to count how many courses a department offers. You can write an SQL function for this:

```

create function course_count(dept_name varchar(20))
returns integer
begin
declare c_count integer;
select count(*) into c_count
from course
where course.dept_name = dept_name;
return c_count;
end;

```

To find out which departments offer more than five courses:

```

select dept_name, budget
from department
where course_count(dept_name) > 5;

```

Similarly, if you want to calculate the total hours an employee has worked, you can define a function like this:

```

create function total_hours_worked(emp_id char(5))
returns integer

```

```

begin
declare hours_worked integer;
select sum(hours) into hours_worked
from timesheet
where timesheet.emp_id = emp_id;
return hours_worked;
end;

```

This function can be used to find employees who have worked more than 40 hours in a week:

```

select emp_name
from employee
where total_hours_worked(emp_id) > 40;

```

## Triggers

**Triggers** are SQL commands that automatically execute when certain actions are performed on a table, such as inserting or updating a record. Triggers are often used to maintain data integrity or to automate business rules.

### *Trigger Example*

Imagine that we want to automatically update a student's total credits when they pass a course. Here's a trigger that accomplishes this:

```

create trigger update_credits after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade is not null and (orow.grade is null or orow.grade <> nrow.grade)
begin atomic
update student
set tot_cred = tot_cred + (select credits from course where course.course_id = nrow.course_id)
where student.id = nrow.id;
end;

```

This trigger checks if the grade has changed from null or "F" to a passing grade and updates the total credits accordingly.

Now, consider an example where we need to apply a bonus automatically when an employee's salary is updated. The following trigger ensures that employees with a salary over \$80,000 receive a 10% bonus:

```

create trigger apply_bonus after update of employee on (salary)
referencing new row as nrow
for each row
when nrow.salary > 80000
begin atomic
update employee
set bonus = salary * 0.1
where employee.id = nrow.id;
end;

```

---

## Recursive Queries

SQL supports **recursive queries**, which are particularly useful when dealing with hierarchical data such as organizational charts or course prerequisites. Recursive queries allow the query to refer to itself, producing results until no further data can be retrieved.

### *Recursive Query for Course Prerequisites*

To find all courses that are prerequisites for another course, you can write a recursive query like this:

```
with recursive prereq_chain(course_id, prereq_id) as (
select course_id, prereq_id from prereq
union
select prereq_chain.course_id, prereq.prereq_id
from prereq_chain, prereq
where prereq_chain.prereq_id = prereq.course_id
)
select * from prereq_chain;
```

This query will return all courses that are directly or indirectly required as prerequisites.

A similar scenario arises when you want to find all employees who report to a certain manager, directly or indirectly. You can write a recursive query to handle this:

```
with recursive emp_hierarchy(emp_id, manager_id) as (
select emp_id, manager_id from employee
where manager_id = 'M001'
union
select employee.emp_id, employee.manager_id
from emp_hierarchy, employee
where emp_hierarchy.emp_id = employee.manager_id
)
select * from emp_hierarchy;
```

---

## Advanced Aggregation Features

SQL's advanced aggregation functions allow for more complex analysis, such as ranking data. Functions like `rank()`, `dense_rank()`, and `percent_rank()` are commonly used for ranking rows based on the values of specific columns.

### *Ranking Example*

To assign a rank to students based on their GPA:

```
select student_id, gpa, rank() over (order by gpa desc) as rank
from student_grades;
```

This query assigns a rank to each student, with the highest GPA receiving the highest rank.

For another use case, you might want to rank employees based on their sales performance:

```
select emp_id, total_sales, rank() over (order by total_sales desc) as sales_rank
from sales_performance;
```

This query will rank employees based on their total sales, with the top seller receiving the highest rank.

---

## Windowing Functions

**Windowing functions** in SQL allow for calculations across sets of rows that are related to the current row. These are useful for tasks like calculating running totals, moving averages, or cumulative sums.

### *Moving Average Example*

To calculate a moving average of sales over time, you can use a window function like this:

```
select date,
avg(sales_amount) over (order by date rows between 1 preceding and 1 following) as moving_avg
from sales_data;
```

This query calculates the average sales value for each date, using the current date, the day before, and the day after.

Similarly, to calculate a moving average of website visitors, you might write:

```
select visit_date,
avg(visits) over (order by visit_date rows between 1 preceding and 1 following) as visit_avg
from website_traffic;
```

---

## OLAP (Online Analytical Processing)

OLAP tools allow for the analysis of large amounts of data by performing multi-dimensional queries. SQL supports OLAP operations with functions like `cube()` and `rollup()` to summarize data across multiple dimensions.

### *OLAP Cube Example*

To analyze sales data across multiple dimensions (item, color, and size), you can use the `cube()` function:

```
select item_name, color, size, sum(quantity_sold)
from sales
group by cube(item_name, color, size);
```

This will generate a summary for every possible combination of item, color, and size.

For summarizing website traffic by device, location, and time, the `cube()` function can be used as follows:



```
select device_type, location, visit_time, sum(visits)
from website_visits
group by cube(device_type, location, visit_time);
```

## Unit: 2

### Database Design Using the E-R Model

The Entity-Relationship (E-R) Model is a foundational tool used in designing databases.

#### Design Phases

The design of a database follows a structured process, generally broken down into three main phases:

1. **Initial Phase:** This stage involves gathering and characterizing the data requirements of the prospective database users. The goal is to fully understand the data needs of the application.
2. **Second Phase:** Here, the focus is on selecting a suitable data model and using it to create a conceptual schema for the database. This schema should reflect the functional requirements of the system, detailing the types of transactions and operations that will be performed on the data.
3. **Final Phase:** This phase translates the abstract schema into an actual database structure. It involves:
  - **Logical Design:** Determining the organization of data into relation schemas, deciding what attributes are essential, and how they should be structured.
  - **Physical Design:** Planning the physical layout of the database on storage media for efficient access and modification.

Effective database design avoids pitfalls such as **redundancy** (repeated information, which may lead to inconsistency) and **incompleteness** (design flaws that make certain functions difficult or impossible to perform).

## Database Design Approaches

There are multiple approaches to designing a database schema:

- **Entity-Relationship (E-R) Model:** The E-R model represents data in terms of entities (distinct objects) and relationships (associations among entities). Entities have attributes that describe their characteristics, while relationships link entities together.
- **Normalization Theory:** This formal approach (covered in more detail in the next section) identifies potentially flawed designs and provides a way to improve them.

The E-R model is illustrated using **E-R diagrams** that visually represent entities, relationships, and constraints.

## Entity Sets

An **entity** represents a real-world object that is distinguishable from other objects. Examples include a person, a company, or an event. An **entity set** is a collection of similar entities, like all employees in a company.

Each entity has **attributes**, or characteristics, that provide more details. For example:

- Employee = (Employee\_ID, Name, Department, Salary)
- Product = (Product\_ID, Name, Price, Category)

In each entity set, a unique identifier or **primary key** is chosen. This key uniquely identifies each entity within the set.

## Representing Entity Sets in E-R Diagrams

In E-R diagrams:

- **Rectangles** represent entity sets.
- **Attributes** are listed inside the rectangle for each entity set, with the **primary key** underlined to indicate its uniqueness.

For example, a Customer entity set might be represented with attributes like Customer\_ID (primary key), Name, and Contact\_Number.

## Relationship Sets

A **relationship** defines an association among two or more entities. For example, the relationship "purchases" could link a Customer and a Product. The **relationship set** contains all such associations, typically between entities from different sets.

Consider the works\_for relationship, which associates employees with their departments. The relationship set might include instances like:

(Employee\_ID\_123, HR), (Employee\_ID\_456, Marketing)

In E-R diagrams, **diamonds** represent relationship sets, and lines link them to the relevant entity sets.

### Attributes in Relationship Sets

Relationships can also have attributes. For instance, the advisor relationship set between Student and Instructor entities could include an attribute like advising\_start\_date, which indicates when the advising relationship began.

For example, consider a relationship set supervises between Manager and Project, with an attribute start\_date that records when supervision began.

### Roles in Relationships

Sometimes, an entity set plays multiple roles in a relationship. For example, in a mentor relationship, an Employee entity may serve as both the mentor and the mentee.

In such cases, **roles** are labeled to clarify each entity's position in the relationship. For example:

- Mentor (role) - Mentee (role)

### Degree of a Relationship Set

Relationships generally involve two entity sets (binary relationships), but occasionally, relationships may involve three or more entity sets (ternary or higher).

For instance, a project\_assignment relationship might involve Employee, Project, and Role entities. This relationship captures which employee works on which project in what capacity.

### Complex Attributes

Attributes can be categorized as:

- **Simple:** Cannot be divided (e.g., age).
- **Composite:** Can be subdivided into parts (e.g., address might include street, city, and postal\_code).
- **Single-valued or Multivalued:** A single-valued attribute has only one value (e.g., SSN), while a multivalued attribute can have multiple values (e.g., phone\_numbers).

Attributes may also be **derived** if they can be calculated from other attributes, such as calculating age based on date\_of\_birth.

### Mapping Cardinality Constraints

**Cardinality constraints** specify the number of entities that can be associated with another entity in a relationship. In binary relationships, there are four possible cardinalities:

1. **One-to-One (1:1)**: An entity in one set is related to at most one entity in another set.
  - Example: Each manager is assigned to at most one department.
2. **One-to-Many (1:N)**: An entity in one set is associated with multiple entities in another set.
  - Example: A department can have many employees, but each employee belongs to only one department.
3. **Many-to-One (N:1)**: Many entities in one set relate to a single entity in another.
  - Example: Many employees report to a single manager.
4. **Many-to-Many (M:M)**: Multiple entities in both sets are associated with each other.
  - Example: Students can enroll in multiple courses, and each course can have multiple students.

### Total and Partial Participation

**Total participation** means every entity in a set is involved in at least one relationship. For instance, every Student might need an Advisor.

**Partial participation** means only some entities participate in the relationship. For instance, only some Employees might be part of a mentorship program.

### Primary Key for Entity and Relationship Sets

A **primary key** is a unique identifier for each entity in a set. In relationship sets, the primary key may combine the primary keys of the participating entities.

For instance, in an assignment relationship between Employee and Project, the primary key may consist of the Employee\_ID and Project\_ID.

### Weak Entity Sets

A **weak entity set** is an entity that cannot be uniquely identified by its own attributes alone. Instead, it depends on an associated **strong entity set**. For example, a Dependent entity may be linked to an Employee, where Dependent lacks unique identifying attributes and is therefore a weak entity.

Weak entities are represented in E-R diagrams with a double rectangle and connected to their identifying entity by a double diamond.

### Redundant Attributes

Redundant attributes replicate information and should generally be avoided. For example, if Department\_Name is already part of a Department entity, including it in an Employee entity as well could lead to inconsistency if department details change.

### Extended E-R Model Features

The E-R model can be enhanced with features such as:

- **Specialization and Generalization:** A **specialization** hierarchy allows for creating subgroups within an entity set, while **generalization** combines entity sets into a broader category.
- **Aggregation:** Aggregation treats a relationship as a higher-level entity, which enables relationships between relationships.

In a specialization, an Employee could be specialized into Manager and Staff categories, each with unique attributes.

---

### Mappings:

In databases, mappings refer to the associations between different schema levels (external, conceptual, and internal) that ensure data consistency across views and allow users to access data without needing to understand underlying storage details.

1. **Data Independence:** This property allows changes to one schema level without affecting others. It enables flexibility and adaptability within the database system.
2. **Logical Data Independence:** Allows changes to the conceptual schema, like adding new fields or tables, without requiring changes to the external schema or application programs. It is harder to achieve than physical data independence.
3. **Physical Data Independence:** Allows changes to the internal schema, such as changes in storage or indexing methods, without affecting the conceptual schema or application programs. It is easier to achieve and helps optimize storage and access.

### Integrity Principles:

Integrity Principles in databases ensure data accuracy, consistency, and reliability. Key principles include:

- **Entity Integrity:** Ensures each table has a primary key, and that primary key values are unique and not null, ensuring each row can be uniquely identified.
- **Referential Integrity:** Maintains consistent relationships between tables by ensuring that foreign key values either match a primary key value in another table or are null, preventing orphan records.
- **Domain Integrity:** Enforces valid data entries by restricting column values to a specific data type, format, or range, preserving data correctness within defined limits.

---

### Normalization

Normalization in relational database design is the process of structuring data to minimize redundancy and improve data integrity. This class covers key principles of normalization, including features of good relational design, types of dependencies, various normal forms, and decomposition strategies.

### Features of Good Relational Design

Good relational design organizes data logically to minimize redundancy and dependency. For example, if we combine an instructor table with a department table, we may end up with a table like `in_dep` below. This can lead to redundant information and null values, especially if new departments are added without instructors:

#### **Instructor\_ID Name Salary Dept\_Name Building Budget**

101	Alice	90000	Physics	Science	500000
-----	-------	-------	---------	---------	--------

**Instructor\_ID Name Salary Dept\_Name Building Budget**

102	Bob	80000	Chemistry	Arts	300000
103	Carol	NULL	Humanities	NULL	NULL

**Decomposition**

Decomposition splits a relation into smaller, more meaningful tables. This process can improve database integrity and reduce redundancy. However, not all decompositions are effective. Consider the employee table below:

**ID Name Street City Salary**

E01	Alice	Maple St.	New York	70000
-----	-------	-----------	----------	-------

E02	Bob	Oak St.	Boston	80000
-----	-----	---------	--------	-------

If we decompose this into:

**Table 1: employee1****ID Name**

E01	Alice
-----	-------

E02	Bob
-----	-----

**Table 2: employee2****Name Street City Salary**

Alice	Maple St.	New York	70000
-------	-----------	----------	-------

Bob	Oak St.	Boston	80000
-----	---------	--------	-------

This decomposition loses the direct relationship between ID and address attributes, leading to ambiguity and potential data loss if two employees share the same name.

### Lossless Decomposition

A **lossless decomposition** allows data to be decomposed without loss of information. If a relation R with attributes (A, B, C) is decomposed into:

**Table 3: R1**

**A B**

A1 B1

A2 B2

**Table 4: R2**

**B C**

B1 C1

B2 C2

If B determines one of the original attributes in R, we can recombine these tables to recreate the original R without data loss, achieving a lossless decomposition.

### Functional Dependencies (FDs)

**Functional dependencies** (FDs) specify that an attribute or set of attributes uniquely determines another attribute. For instance:

#### Instructor Table

**Student\_ID Name Dept\_Name**

S01 Alice Physics

S02 Bob Chemistry

Here, Student\_ID uniquely identifies each student, represented by Student\_ID  $\rightarrow$  Name. Similarly, if Dept\_Name determines the Building in a Department table, we have Dept\_Name  $\rightarrow$  Building.

### Trivial Functional Dependencies

A **trivial dependency** always holds if  $\beta$  is a subset of  $\alpha$ . For example:



**ID Name**

101 Ann

102 Bob

Since ID determines itself, we have  $ID, Name \rightarrow ID$ , a trivial dependency.

**Keys and Functional Dependencies**

A **superkey** uniquely identifies each tuple in a relation, and a **candidate key** is a minimal superkey. Consider the `in_dep` schema with attributes (ID, name, salary, dept\_name, building, budget), where the following dependencies might hold:

- $Dept\_Name \rightarrow Building$
- $ID \rightarrow Dept\_Name$

This structure makes ID a candidate key for identifying each instructor uniquely.

**Boyce-Codd Normal Form (BCNF)**

A relation is in **BCNF** if, for every FD  $\alpha \rightarrow \beta$ , at least one of the following holds:

1.  $\alpha \rightarrow \beta$  is trivial.
2.  $\alpha$  is a superkey.

For example, consider `in_dep` with attributes (ID, name, salary, dept\_name, building, budget) and dependencies:

- $Dept\_Name \rightarrow Building, Budget$

To achieve BCNF, we decompose the table into instructor and department as follows:

**Instructor Table****ID Name Salary Dept\_Name**

101 Alice 90000 Physics

102 Bob 80000 Chemistry

**Department Table**

**Dept\_Name Building Budget**

Physics Science 500000

Chemistry Arts 300000

**Third Normal Form (3NF)**

**Third Normal Form (3NF)** allows dependencies where a non-superkey attribute in  $\beta$  is part of a candidate key. Consider the following dept\_advisor schema:

**Dept\_Advisor Table**

**s\_ID i\_ID Dept\_Name**

S01 I101 Physics

S02 I102 Chemistry

If  $i\_ID \rightarrow Dept\_Name$  and  $s\_ID, Dept\_Name \rightarrow i\_ID$ , this relation is in 3NF but not necessarily in BCNF.

**Multivalued Dependencies (MVDs)**

**Multivalued dependencies** arise when one attribute determines multiple independent values in another. For instance, an inst\_info table with (ID, child\_name, phone\_number) can be decomposed as follows to avoid redundancy:

**Inst\_Child Table**

**ID Child\_Name**

101 Emma

101 Ryan

**Inst\_Phone Table**

**ID Phone\_Number**

101 555-1234

101 555-5678

Here, each child's name and phone number are stored independently, avoiding redundancy.

### Fourth Normal Form (4NF)

A relation is in **4NF** if it is in BCNF and has no non-trivial multivalued dependencies. By decomposing `inst_info` with attributes (ID, child\_name, phone\_number) as shown above, we avoid redundancy and achieve 4NF.

### Dependency Preservation and Canonical Cover

**Dependency preservation** ensures that FDs can be enforced on individual tables without reconstructing the original relation. To achieve this, we use a **canonical cover** to simplify FDs without losing dependency information. Consider the following FDs:

1.  $A \rightarrow BC$
2.  $B \rightarrow C$
3.  $A \rightarrow B$

The canonical cover simplifies these to:

- $A \rightarrow B$
- $B \rightarrow C$

Ensuring dependency preservation helps maintain data integrity and efficiency in database operations.

---

## ADVANCED TOPICS

### Big Data

With the massive amounts of data generated by web usage, social media, and the Internet of Things (IoT), handling and processing such large datasets have become a challenge.

#### 1. Motivation for Big Data

As the volume of data increases, traditional databases struggle to handle the following three main characteristics of big data:

- **Volume:** The sheer size of the data has grown exponentially.
- **Velocity:** Data is generated and needs to be processed at an increasingly high speed.
- **Variety:** There are diverse types of data, such as structured, semi-structured, and unstructured data (e.g., web logs, social media posts, sensor data).

Analyzing big data is critical for optimizing advertisements, structuring web pages, and personalizing user content.

## 2. Querying Big Data

In environments that deal with massive datasets, the traditional **ACID** properties of databases (Atomicity, Consistency, Isolation, Durability) are often sacrificed for scalability and speed. Querying big data requires scalable systems that can support non-relational data and high-throughput query processing.

## 3. Big Data Storage Systems

Several approaches are used for storing big data efficiently:

- **Distributed File Systems:** These store data across many machines while giving the appearance of a single file system. Popular examples include **Google File System (GFS)** and **Hadoop Distributed File System (HDFS)**.
- **Sharding:** This partitions data across multiple databases based on a partition key.
- **Key-Value Storage Systems:** These systems store data as key-value pairs, allowing fast access to records across multiple machines.
- **Parallel and Distributed Databases:** These databases run across multiple machines, splitting data and queries for parallel processing.

## 4. Distributed File Systems

A **distributed file system (DFS)** stores data across many machines but provides a unified file system view. These systems are highly scalable and provide fault tolerance by replicating data across nodes.

- **Example:** Google File System (GFS) and Hadoop Distributed File System (HDFS) are two examples where data is split into blocks (e.g., 64 MB each) and replicated across multiple nodes to ensure reliability.

## 5. Hadoop File System Architecture

The Hadoop Distributed File System (HDFS) divides files into blocks, typically 64 MB in size, and replicates these blocks across several machines for fault tolerance. The architecture involves:

- **NameNode:** Manages the metadata, such as mapping file names to block IDs and blocks to the physical locations.
- **DataNode:** Stores actual data blocks and serves them to clients.

Data is accessed by clients directly from the DataNodes, with the NameNode providing information on where blocks are located.

## 6. Sharding

**Sharding** is the process of partitioning data across multiple databases. Data is divided into "shards" based on a **shard key** such as user\_ID. For example:

- Data with key values between 1 and 100,000 might be stored on one database, while values between 100,001 and 200,000 are stored on another.

While sharding offers scalability, it requires applications to track which database holds which portion of the data, increasing complexity.

## 7. Key-Value Storage Systems

Key-value storage systems handle massive datasets by storing records as key-value pairs and distributing them across multiple machines.

- **Examples:**
  - **Amazon S3** stores objects as large files with associated metadata.
  - **Google BigTable** and **Apache Cassandra** offer a wide-table format, allowing many attribute names to be associated with each key.

These systems ensure availability and consistency by replicating data across machines.

## 8. Data Representation

Semi-structured data formats like **JSON** and **XML** are popular for representing key-value data in NoSQL databases. JSON allows data to be stored in flexible schemas, supporting various complex data structures.

- **Example of JSON:**

```
json
{
  "ID": "22222",
  "name": {
    "firstname": "Albert",
    "lastname": "Einstein"
  },
  "deptname": "Physics",
  "children": [
    {"firstname": "Hans", "lastname": "Einstein"},
    {"firstname": "Eduard", "lastname": "Einstein"}
  ]
}
```

## 9. Parallel and Distributed Databases

Parallel databases distribute their workload across multiple machines (a cluster) to process data faster. These databases ensure data availability by replicating data but face challenges such as frequent query restarts when machines fail.

MapReduce systems, such as Hadoop, work around failures by continuing the processing on functioning machines without needing to restart the entire query.

## 10. Replication and Consistency

To ensure availability, distributed databases often replicate data across multiple nodes. This ensures that if one machine fails, another can continue serving requests. However, replication introduces challenges with consistency:

- **Consistency:** Ensures that all replicas contain the same data, and all reads reflect the latest updates.
- **Availability:** Guarantees that the system remains operational even when parts of it fail.

**CAP Theorem** states that a distributed system can guarantee only two of the following three: Consistency, Availability, and Partition Tolerance (i.e., the system remains operational despite network partitions).

---

## 11. MapReduce Paradigm

**MapReduce** is a programming model used for processing large datasets across many machines. It abstracts the complexities of distributed and parallel processing, allowing developers to focus on the logic of their computations.

The **MapReduce process** involves:

1. **Map Function:** Takes an input, processes it, and produces key-value pairs.
  2. **Reduce Function:** Groups these pairs and processes them to provide a final output.
- **Example:** Counting the number of occurrences of each word in a collection of documents. The map function outputs word-count pairs, and the reduce function aggregates these counts by word.

## 12. Hadoop and MapReduce

Hadoop is an open-source implementation of MapReduce, widely used for processing large datasets. It uses **HDFS** for data storage, and map and reduce functions can be written in several programming languages (e.g., Java, Python).

### 13. Streaming Data

**Streaming data** refers to continuous data flow generated in real-time from various sources such as stock markets, IoT sensors, and social media platforms. Stream processing systems allow us to run continuous queries on this data to monitor and trigger actions in real-time.

- **Windowing:** Breaks up a continuous stream of data into smaller windows (e.g., hourly or session-based windows) for processing.
- **Pattern Matching:** Detects specific patterns in data streams and triggers actions.

### 14. Graph Databases

**Graph databases** model data in terms of nodes (entities) and edges (relationships). This model is highly efficient for representing and querying complex, interconnected datasets such as social networks or recommendation systems.

- **Example:** In a social network graph, nodes represent users, and edges represent "friend" or "follower" relationships.

**Graph Query Languages** like **Neo4J** allow for easy traversal of nodes and relationships, making it simpler to execute complex queries, such as finding all friends-of-friends in a social network.

### 15. Bulk Synchronous Processing (BSP)

**Bulk Synchronous Processing (BSP)** is a framework used for processing large graphs in parallel. It divides computation into multiple **supersteps**, where each node in the graph sends and receives messages from its neighbors, updating its state iteratively. The process continues until all nodes complete their tasks.

- **Example:** Google's **Pregel** and Apache's **Giraph** are based on the BSP framework, enabling efficient parallel processing of graphs with billions of nodes and edges.

## Data Analytics

Data analytics involves processing data to uncover patterns, correlations, and predictive models. It is vital for making informed business decisions, from personalized customer suggestions to high-level stock management strategies.

### 1. Data Warehousing

A **data warehouse** serves as a central repository of information, consolidating data from multiple sources and storing it under a unified schema. Unlike operational databases, data warehouses store historical data, providing a foundation for complex decision-making processes.

- **Example:** A retail company may gather data from sales, inventory, and customer feedback into a warehouse to analyze purchasing trends and stock requirements.

### Design Considerations

1. **Source-Driven Architecture:** Data sources transmit updates to the warehouse periodically.
2. **Destination-Driven Architecture:** The warehouse requests updates from sources.
3. **Synchronous vs. Asynchronous Replication:** While real-time synchronization is ideal, asynchronous replication often balances performance with minor latency.
4. **Schema Integration:** A unified schema helps in consistent querying and analysis across datasets from diverse sources.

## 2. Multidimensional Data and Warehouse Schemas

Data in a warehouse often uses a multidimensional model to structure information as **fact** and **dimension tables**.

- **Fact Table Example:** The sales fact table might include fields like `item_id`, `store_id`, `customer_id`, `date`, `quantity`, and `price`.
- **Dimension Table Example:** The store dimension table could include details such as `store_id`, `location`, and `manager_name`.

A common schema type is the **star schema**, where a central fact table is surrounded by dimension tables. More complex structures, such as the **snowflake schema**, further decompose dimension tables.

## 3. Online Analytical Processing (OLAP)

OLAP enables interactive, multi-dimensional analysis, allowing users to perform aggregations, comparisons, and other calculations on large datasets.

- **OLAP Operations:**
  - **Pivoting:** Adjusts the displayed dimensions in a cross-tab.
  - **Slicing:** Filters the dataset for specific values within a dimension.
  - **Roll-Up:** Aggregates data to a higher level (e.g., daily sales aggregated to monthly totals).
  - **Drill-Down:** Explores data at a finer level (e.g., breaking down monthly totals into daily figures).

### Example OLAP Query

Consider a sales table with fields for `item_name`, `color`, `size`, and `quantity`. An OLAP query might compute the total quantity sold by item and color, forming a cross-tab:

#### Item Name Dark Pastel White

Shirt	150	200	50
Pants	100	120	90



## 4. Data Mining

Data mining is the automated analysis of large datasets to find meaningful patterns. These patterns often form the basis of predictive models or descriptive statistics that businesses can use for decision-making.

### *Types of Data Mining Tasks*

1. **Prediction:** Uses past data to forecast outcomes.
  - **Example:** Predicting customer credit risk based on attributes like income, job type, and age.
2. **Classification:** Assigns items to predefined classes.
  - **Example:** Classifying emails as "spam" or "not spam" based on keywords and sender details.
3. **Clustering:** Groups similar items.
  - **Example:** Grouping customers with similar purchase histories to design targeted marketing campaigns.
4. **Association Rules:** Finds relationships among data items.
  - **Example:** If a customer buys bread, they are likely to also buy milk.

## 5. Decision Trees

**Decision trees** classify data by partitioning it into subsets based on attribute values. Each node in the tree represents a test on an attribute, and branches represent possible outcomes. A decision tree's leaf nodes represent classes.

- **Example:** A decision tree predicting loan default based on income, employment\_status, and credit\_score might have nodes where:
  - If income > 50k and credit\_score > 700, then No Default.
  - If income <= 50k and employment\_status = Part-time, then Potential Default.

## 6. Bayesian Classifiers

**Bayesian classifiers** apply Bayes' theorem to calculate the probability of an outcome based on observed data.

- **Bayes' Theorem:**  $P(C|D) = \frac{P(D|C) \cdot P(C)}{P(D)}$  where:
  - $P(C|D)P(D)$  is the probability of class CCC given data DDD.
  - $P(D|C)P(C)$  is the probability of observing DDD if CCC is true.

In practice, **Naïve Bayes** simplifies this calculation by assuming attribute independence, making it computationally feasible for large datasets.

## 7. Support Vector Machine (SVM) Classifiers

SVMs are used for binary classification, finding a boundary that best separates data points from different classes. In 2D space, this boundary is a line; in higher dimensions, it's a hyperplane.

- **Example:** To classify emails as "spam" or "not spam," SVMs identify an optimal dividing line that separates spam emails (based on words like "free" or "prize") from legitimate emails.

## 8. Neural Networks

Neural networks are computational models inspired by the human brain, composed of layers of interconnected nodes (neurons) that process data.

### *Components of a Neural Network*

1. **Input Layer:** Receives raw data.
  2. **Hidden Layers:** Perform calculations, learning patterns in the data.
  3. **Output Layer:** Produces predictions.
- **Example:** For image recognition, a neural network can classify an image as a "cat" or "dog" by analyzing pixel patterns and adjusting weights through training.

## 9. Regression Analysis

**Regression** aims to predict a continuous value rather than classify an instance. Given values for variables  $X_1, X_2, \dots, X_n$ , regression predicts an outcome  $Y$  using a formula like:

$$Y = a_0 + a_1 \cdot X_1 + a_2 \cdot X_2 + \dots + a_n \cdot X_n$$

- **Example:** Predicting house prices based on features like square footage, neighborhood, and age.

## 10. Association Rules

**Association rules** reveal relationships between items in a dataset, such as frequent itemsets in transaction data.

- **Support:** Indicates the proportion of the population that satisfies both antecedent and consequent.
- **Confidence:** Measures how often the consequent is true when the antecedent is true.

### *Example*

An online bookstore may find that customers who buy "Database Systems" are also likely to buy "Operating System Concepts." This association can then be used to recommend books during checkout.

## 11. Clustering

**Clustering** groups data points such that items in the same group are more similar to each other than to those in other groups.

- **Example:** Clustering students by study habits can identify groups needing different levels of academic support.

**Hierarchical Clustering** builds clusters at various levels, with techniques like **agglomerative clustering** (bottom-up) and **divisive clustering** (top-down).

## 12. Text Mining and Sentiment Analysis

**Text mining** applies data mining to textual content, while **sentiment analysis** determines the emotional tone in texts, such as reviews or social media posts.

- **Example:** A sentiment analysis of customer reviews on a product can help gauge overall customer satisfaction.

## Physical Storage Systems

This part explores the physical storage mechanisms that underlie modern database systems, covering topics such as storage media types, the storage hierarchy, disk and flash storage mechanisms, RAID (Redundant Arrays of Independent Disks), and hardware considerations for data integrity and reliability.

### 1. Classification of Physical Storage Media

Storage media are classified based on data volatility and performance characteristics:

1. **Volatile Storage:** Loses data when power is turned off.
  - **Example:** Cache memory and main memory.
2. **Non-Volatile Storage:** Retains data without power, encompassing secondary and tertiary storage.
  - **Example:** Hard disks and optical storage.

Factors influencing storage media selection include **speed** (data access rates), **cost per unit of data**, and **reliability**.

### 2. Storage Hierarchy

The storage hierarchy arranges storage types by speed and cost:

- **Primary Storage:** Fast, volatile storage like cache and main memory.
- **Secondary Storage:** Non-volatile storage with moderate speed, e.g., flash memory and magnetic disks.
- **Tertiary Storage:** Slow, archival storage, often offline, such as magnetic tape.

Each tier has a specific role, balancing data access speed and storage costs.

### 3. Storage Interfaces

There are several standards for connecting storage devices:

- **SATA (Serial ATA):** Supports data transfer speeds up to 6 Gbps.
- **SAS (Serial Attached SCSI):** Supports up to 12 Gbps.
- **NVMe (Non-Volatile Memory Express):** Works with PCIe to achieve transfer rates up to 24 Gbps, suitable for high-performance flash storage.

### 4. Magnetic Hard Disk Mechanism

A **magnetic disk** is organized into circular tracks on a spinning platter, divided into sectors, which are the smallest data units that can be read or written.

- **Tracks and Sectors:**
  - Each platter holds 50,000 to 100,000 tracks.
  - Sectors are typically 512 bytes, with inner tracks holding 500 to 1,000 sectors and outer tracks up to 2,000.

Data access involves positioning the read-write head over the appropriate track as the disk spins. Multiple platters may be mounted on a single spindle, and each track on the platters forms a **cylinder**.

### 5. Disk Performance Measures

Performance of disks is typically measured by:

- **Seek Time:** The time taken to position the head over the correct track. Average seek time ranges from 4 to 10 ms.
- **Rotational Latency:** The wait for the desired sector to pass under the head, averaging 4 to 11 ms.
- **Data Transfer Rate:** Ranges from 25 to 200 MB/s, varying with disk model and track location.

### 6. Flash Storage

**Flash memory** is a type of non-volatile storage with two main types:

- **NOR Flash:** Common in low-capacity, high-speed reads.

- **NAND Flash:** Used in high-capacity SSDs, read speeds vary from 20 to 100 microseconds for a 512-byte page.

Flash requires pages to be erased before rewriting, typically in blocks of 256 KB to 1 MB. A **Flash Translation Table (FTT)** maps logical to physical page addresses, enabling efficient data retrieval and wear leveling.

## 7. SSD Performance Metrics

Solid-State Drives (SSDs) support high input/output operations per second (IOPS):

- **Reads:** 100,000 IOPS with 32 parallel requests on SATA and up to 350,000 on NVMe.
- **Writes:** 100,000 IOPS on high-end models.

SSDs achieve data transfer rates of up to 3 GB/s on NVMe, with hybrid drives combining flash cache and magnetic storage for better performance.

## 8. Storage Class Memory

**Storage Class Memory (SCM)** bridges the gap between SSDs and DRAM, offering faster access than traditional SSDs. Intel's **3D XPoint** technology, marketed as Intel Optane, is an SCM that provides high-speed, non-volatile storage.

## 9. RAID: Redundant Arrays of Independent Disks

RAID organizes multiple disks into arrays, enhancing storage capacity, speed, and reliability through redundancy.

### *RAID Levels:*

- **RAID 0:** Block-level striping without redundancy, used for high-speed applications where data loss is acceptable.
- **RAID 1:** Mirrored disks with block striping, offering high reliability and fast reads.
- **RAID 5:** Block-interleaved distributed parity, distributing data and parity across disks, improving performance.
- **RAID 6:** Similar to RAID 5 but includes two parity blocks for enhanced protection against multiple disk failures.

Each RAID level has specific use cases depending on cost, performance, and reliability requirements.

## 10. Hardware Issues

RAID systems can be implemented in two ways:

- **Software RAID:** Managed by the operating system without special hardware.
- **Hardware RAID:** Uses dedicated hardware with features like non-volatile RAM for tracking in-progress writes, reducing the risk of corruption from power loss.

**Hot Swapping** allows disks to be replaced without shutting down the system, maintaining availability.

---

## 11. Optimization of Disk-Block Access

Optimizing disk access involves techniques such as:

- **Buffering:** Uses in-memory buffers to store frequently accessed data.
- **Read-Ahead:** Reads additional blocks in anticipation of upcoming requests.
- **Disk-Arm Scheduling:** Reduces arm movement to minimize access time, using algorithms like the **elevator algorithm** to sequence requests efficiently.

## Magnetic Tapes

Magnetic tapes offer high-capacity storage, primarily used for backup and archival purposes. While tapes hold large volumes (up to several petabytes), they are limited to sequential access, making them unsuitable for frequent, random access but effective for low-cost data storage.

## Data Storage Structures

Database storage structures organize data for efficient storage and retrieval, enhancing database performance and reliability. This class covers file organization, record structures, clustering techniques, and memory management methods.

### 1. File Organization

In databases, data is stored in files, each containing records made up of fields. Here, files can be organized by record type and size:

- **Fixed-size records:** Simple files containing records of one specific type with a fixed size.
- **Multiple Files for Different Relations:** Separate files store records for different tables or entities.

**Example:** An Employee table with fixed-size records might look like this:

**Employee\_ID Name Department Salary**

001	Alice	HR	60000
002	Bob	IT	75000
003	Carol	Marketing	67000

**2. Fixed-Length Records**

**Fixed-length records** have a set amount of space. When records are deleted, several methods can manage the vacant space:

- **Shift records** to fill the gap.
- **Reposition** the last record in the file to the deleted spot.
- Use a **free list** to manage empty slots for reuse.

**Example:** After deleting record 002 (Bob), the updated file might look like this:

**Employee\_ID Name Department Salary**

001	Alice	HR	60000
(empty)	(empty)	(empty)	(empty)
003	Carol	Marketing	67000

**3. Variable-Length Records**

Variable-length records allow records to have different sizes, handling fields like varying text lengths and multivalued attributes.

A **slotted page structure** can manage these records, storing details like the number of records, the end of free space, and the location of each record.

**Example:** An Employee table with variable-length addresses could look like this:

**Employee\_ID Name Address**

001	Alice	123 Maple St., Apt 10
002	Bob	45 Oak St.
003	Carol	12 Pine Ln., Suite 305

#### 4. Storing Large Objects

For large objects (LOBs) like images or videos, databases can:

1. Store them externally as files.
2. Split them into smaller pieces across multiple records.

**Example:** In PostgreSQL's **TOAST** method, a document might be divided into chunks of 1 KB each and stored in multiple rows.

#### 5. Organization of Records in Files

Different record organizations in a file serve various purposes:

- **Heap Organization:** Records are placed in any available spot.
- **Sequential Organization:** Stores records in a sorted order by a search key.
- **Multitable Clustering:** Stores related records from different tables together.

**Example:** For a Student-Course clustering:

**Student\_ID Name Course\_Code Course\_Name**

S001	Alice	C101	Database
S002	Bob	C102	Programming
S001	Alice	C103	Algorithms

#### 6. Heap File Organization

In **heap file organization**, records are stored in the first available slot, making insertion flexible and efficient.

A **free-space map** tracks free blocks, allowing efficient space allocation for new records.

**Example:** For an Inventory table:

**Item\_ID Description Quantity Location**

1001	Monitor	30	A1
1002	Keyboard	15	B3
(empty)	(empty)	(empty)	(empty)



## 7. Sequential File Organization

Sequential file organization suits tasks requiring ordered data processing.

For insertions in sorted sequences, **overflow blocks** may store additional records, keeping main records in order.

**Example:** An ordered Customer file with an overflow block might look like this:

### Customer\_ID Name Location

C001 Alice NY

C002 Bob CA

Overflow: Dan TX

## 8. Multitable Clustering

With **multitable clustering**, related records from multiple tables are stored in one file for faster data retrieval.

**Example:** Clustering Instructor and Department records:

### Instructor\_ID Name Dept\_ID Dept\_Name

I001 Smith D001 Science

I002 Jones D002 Mathematics

I003 Doe D001 Science

## 9. Partitioning

Partitioning divides a table into smaller segments, allowing storage across devices and optimized access for recent data.

**Example:** A Sales table partitioned by year:

### Year Partition

2021 sales\_2021

2022 sales\_2022

Querying 2022 sales only accesses sales\_2022, enhancing query speed.

## 10. Data Dictionary Storage

A **data dictionary** stores database metadata, containing information on tables, attributes, storage structures, and access privileges.

**Example:** Data dictionary entry for a Customer table:

Attribute	Type	Length	Indexed
Customer_ID	INT	4	Yes
Name	VARCHAR	50	No
Email	VARCHAR	100	Yes

## 11. Buffer Management

The **buffer manager** uses main memory to store disk blocks temporarily, optimizing access.

**Pinned Blocks** remain in memory during active operations, while **replacement policies** (like **LRU** or **MRU**) manage block retention.

**Example:** A query on a large table may use **LRU** to retain frequently accessed blocks and remove others.

## 12. Buffer Replacement Policies

**Replacement policies** determine which blocks stay in memory. Key policies include:

- **LRU (Least Recently Used):** Removes the oldest accessed block.
- **MRU (Most Recently Used):** Retains recent blocks.
- **Toss-Immediate:** Discards blocks after use.

**Example:** For a join operation, **MRU** might retain blocks from both tables if repeatedly accessed.

## 13. Optimization of Disk Block Access

Disk access optimizations include:

- **Forced Output:** Writes blocks to disk to ensure data recovery.
- **Nonvolatile Buffers:** Temporarily store blocks to reduce disk writes.
- **Log Disk:** Sequentially logs updates to minimize disk movement.

**Example:** Log-based storage systems save update logs to prevent scattered writes.

#### 14. Column-Oriented Storage

**Columnar storage** stores each attribute in a separate file, ideal for analytical queries focused on specific columns.

**Example:** In a columnar sales database:

Date	Product	Quantity
------	---------	----------

2023-01-01	Monitor	20
------------	---------	----

2023-01-02	Keyboard	15
------------	----------	----

Only the Quantity column is accessed if analyzing quantities, improving query speed.

#### 15. Columnar File Representation

**ORC** and **Parquet** file formats store data in columns rather than rows, used in big data for compression and speed.

**Example:** A Parquet file for a customer's shopping details might store columns like `transaction_id`, `customer_id`, and `amount` separately.

#### 16. Storage Organization in Main-Memory Databases

**Main-memory databases** store data directly in RAM, speeding up access.

**Example:** An in-memory analytics database might store sales transactions as columns in RAM, enabling real-time analysis.

### Indexing

Indexing is a database mechanism that improves the efficiency of data retrieval. An index allows the database to find and access data faster than scanning every record in a table. This class covers fundamental indexing concepts, ordered indices, B+ trees, hashing, write-optimized indices, and spatial and temporal indexing.

## 1. Basic Concepts of Indexing

An **index** is created to speed up access to data. For instance, a library's author catalog is an index of books by author, enabling efficient lookups.

- **Search Key:** Attribute or set of attributes used to look up records.
- **Index Entry:** Consists of a search key and a pointer to the record.

Index files are smaller than original files, with two primary types:

- **Ordered Indices:** Stores search keys in sorted order.
- **Hash Indices:** Distributes search keys across "buckets" based on a hash function.

## 2. Index Evaluation Metrics

Metrics to assess indexing efficiency include:

- **Access Types:** Ability to retrieve records with specific attributes or within a value range.
- **Access Time:** Speed of retrieving data.
- **Insertion and Deletion Times:** Time required to add or remove records.
- **Space Overhead:** Amount of storage used by the index.

## 3. Ordered Indices

**Ordered indices** maintain index entries sorted by search key values.

- **Clustering (Primary) Index:** The search key specifies the order of the file, typically matching the primary key.
- **Secondary Index:** Uses a search key different from the file's sequential order, known as a non-clustering index.
- **Index-Sequential File:** A sequential file ordered by a search key, with a primary index on that key.

## 4. Dense Index Files

In a **dense index**, every search-key value appears in the index. This index type requires more space but allows faster record access.

**Example:** For an instructor table indexed by ID, each instructor ID has a corresponding index entry.

**ID Name Dept**

101 Alice Math

**ID Name Dept**

102 Bob Science

### 5. Sparse Index Files

A **sparse index** includes entries only for some search-key values, useful for files ordered by the search key.

**Sparse Index Access:**

1. Locate the index record with the largest search key less than K.
2. Search the file sequentially from this point.

Sparse indices use less space but may increase access time.

### 6. Multilevel Index

When an index doesn't fit into memory, a **multilevel index** helps by treating the index as a file and constructing a sparse index on it.

- **Outer Index:** A sparse index of the basic index.
- **Inner Index:** The primary index file.

This hierarchy may have multiple levels if necessary, though all levels must be updated upon insertions or deletions.

### 7. Composite Keys

Composite indices use multiple attributes as search keys.

**Example:** An instructor index on (name, ID) allows sorting by both attributes lexicographically. (John, 12121) is less than (John, 13514), which is less than (Peter, 11223).

### 8. B+-Tree Index Files

**B+-trees** are balanced tree structures widely used for indexing. Key properties include:

- All paths from root to leaf are the same length.
- Each node, except for the root, has between  $\lceil n/2 \rceil$  and n children.
- Leaf nodes contain search keys and pointers to records or buckets.

**B+-Tree Node Structure:**

- Each node holds search keys (K1, K2,...) in sorted order.
- Pointers (P1, P2,...) link to children nodes or records.

**Example:** A B+-tree for a 6-node structure maintains each node's keys in order, balancing access paths and ensuring efficient search, insertion, and deletion operations.

## 9. Hashing

Hashing distributes records across buckets based on a **hash function**. It maps search-key values to bucket addresses, allowing efficient access, insertion, and deletion.

### *Static Hashing*

In **static hashing**, each bucket address corresponds to a fixed set of search-key values.

**Example:** For an instructor table hashed by dept\_name, instructors are grouped by department, allowing fast retrieval based on department names.

## 10. Bucket Overflow and Chaining

**Bucket Overflow:** Occurs when multiple records are hashed to the same bucket. **Overflow Chaining:** Manages overflow by linking additional buckets, forming a linked list.

## 11. Dynamic Hashing

Unlike static hashing, **dynamic hashing** allows bucket numbers to grow or shrink:

- **Linear Hashing:** Expands incrementally by rehashing records in overflowed buckets.
- **Extendable Hashing:** Increases the hash table size without creating new buckets by sharing buckets among hash values.

## 12. Comparison of Ordered Indexing and Hashing

Considerations for selecting between ordered indices and hashing include:

- **Cost:** Hashing may require fewer reorganizations than ordered indexing.
- **Access Type:** Hashing optimizes retrieval of specific values; ordered indices suit range queries.

### 13. Bitmap Indices

**Bitmap indices** store an array of bits for each attribute value, useful when attributes have limited distinct values.

**Example of Bitmap Index on Gender:**

- **Male:** 10010
- **Female:** 01101

Combining bitmaps with **AND** or **OR** operations enables efficient multi-attribute queries.

### 14. Spatial and Temporal Indices

Databases can store and query spatial data (like points, lines, polygons) and temporal data (data with time intervals).

- **k-d Trees:** Used for spatial data by dividing space iteratively along dimensions.
- **Quadtrees:** Each node represents a region, dividing it into four quadrants.
- **R-Trees:** Useful for complex, multi-dimensional data by generalizing B+-trees for bounding boxes.

#### *Example R-Tree*

An R-tree may index geometric shapes like rectangles by bounding them in minimal rectangles, enabling spatial queries like "find all objects within this area."

**Temporal Indexing:** Uses a time dimension to store data intervals, often employing an R-tree for managing spatial-temporal data.

---

## Transaction

Transactions are fundamental units of operation in database systems, involving a series of steps that read and update data to ensure consistency, isolation, and reliability. This class covers transaction concepts, transaction states, the ACID properties, serializability, concurrency control, and SQL transaction handling.

### 1. Transaction Concept

A **transaction** is a logical unit of program execution that accesses and possibly updates various data items. For example, a transaction transferring \$50 from account A to account B involves:

1. read(A)
2. A := A - 50
3. write(A)
4. read(B)

5.  $B := B + 50$
6. write(B)

Key issues include:

- **Failure Handling:** Ensures data consistency despite system or hardware failures.
- **Concurrency Control:** Manages simultaneous transactions without conflict.

## 2. ACID Properties

The **ACID properties** of transactions are essential for data integrity and system reliability:

- **Atomicity:** Ensures that all operations in a transaction are fully completed or have no effect if interrupted. If the above transfer fails after updating A but before updating B, the transaction should revert A's balance to maintain integrity.
- **Consistency:** Enforces database integrity rules, such as primary and foreign key constraints, ensuring each transaction starts and ends with a consistent database.
- **Isolation:** Transactions operate as though they're isolated; intermediate steps in one transaction are invisible to others.
- **Durability:** Guarantees that once a transaction is completed, its effects persist despite subsequent failures.

Example: In a banking system, the ACID properties ensure that transfers are accurately reflected without loss, duplication, or partial processing.

## 3. Transaction States

A transaction can be in one of five states:

1. **Active:** The transaction is executing.
2. **Partially Committed:** The final statement has executed but may not be saved.
3. **Failed:** An error prevents completion.
4. **Aborted:** After a rollback, returning the database to its initial state before the transaction began.
5. **Committed:** Successfully completed, with all changes saved.

## 4. Concurrent Executions

Executing multiple transactions concurrently improves system efficiency by:

- Increasing CPU and disk utilization, as transactions utilize different resources.
- Reducing response time for short transactions by allowing non-blocking concurrent processing.

For example, a transaction calculating daily sales totals can run alongside individual transaction processing without waiting for completion.



## 5. Schedules and Serializability

A **schedule** defines the order of instruction execution among concurrent transactions. For correctness, schedules must preserve the logical order within each transaction.

### *Serializability*

A **serializable schedule** ensures a consistent outcome as though transactions executed sequentially, with no interleaving.

Example of a **serial schedule**:

- T1T\_1T1: Transfer \$50 from A to B.
- T2T\_2T2: Transfer 10% of A's balance to B.

Schedules can be **conflict serializable** (achieved by rearranging non-conflicting operations) or **view serializable** (based on equivalent reads and writes).

---

## 6. Conflict Serializability

**Conflict serializability** arises when schedules can be transformed into serial equivalents by swapping non-conflicting instructions.

### *Conflicting Instructions*

Instructions **conflict** if they access the same data item and at least one is a write. Examples include:

- read(Q) followed by write(Q) (conflict).
- write(Q) followed by write(Q) (conflict).

A conflict serializable schedule maintains consistency by preserving these dependencies.

## 7. View Serializability

**View serializability** ensures two schedules yield the same outcome by reading and writing data consistently. View serializability requirements include:

1. Initial value reads must be preserved.
2. Subsequent reads must match data writes from the same transaction.
3. The final write in each transaction must align with serial scheduling.

**Example:** A schedule where transactions read each other's updates without conflicting can still achieve view serializability.

## 8. Testing for Serializability

Testing involves creating a **precedence graph** with transactions as vertices. An arc is added from  $T_iT_i$  to  $T_jT_j$  if  $T_iT_i$  accesses a data item before  $T_jT_j$  in a conflicting operation.

### *Conflict Serializability Test*

A schedule is conflict serializable if its precedence graph is **acyclic**. Topologically sorting the graph provides a valid serial order.

## 9. Recoverable and Cascadeless Schedules

**Recoverable Schedules:** A schedule is recoverable if a transaction committing after another depends on its completion. If a transaction is aborted, others depending on its results can also roll back.

Example of cascading rollback:

- $T_{1T_1}$ : write(X) followed by  $T_{2T_2}$ : read(X) means if  $T_{1T_1}$  fails,  $T_{2T_2}$  must also roll back.

**Cascadeless Schedules** avoid cascading rollbacks by ensuring a transaction only reads committed data.

---

## 10. Concurrency Control

Concurrency control mechanisms ensure schedules are:

1. Conflict or view serializable.
2. Recoverable and, ideally, cascadeless.

Techniques vary between **pessimistic locking** (ensuring exclusive access) and **optimistic methods** (detecting and managing conflicts after they occur).

### *Example of Locking*

In a two-phase locking (2PL) protocol, transactions lock data items, preventing conflicts until they're ready to commit, at which point all locks are released.

## 11. SQL Transaction Commands

Transactions in SQL can be controlled using:

- **BEGIN TRANSACTION:** Marks the start.
- **COMMIT:** Saves changes and ends the transaction.
- **ROLLBACK:** Reverts all changes.

SQL supports setting **isolation levels**:

- **Serializable**: Highest isolation, preventing other transactions from accessing uncommitted data.
- **Read Committed**: Reads only committed data but allows changes by others during a transaction.
- **Read Uncommitted**: Allows reading uncommitted data.

Example:

sql

```
BEGIN TRANSACTION;  
UPDATE accounts SET balance = balance - 100 WHERE account_id = 'A';  
UPDATE accounts SET balance = balance + 100 WHERE account_id = 'B';  
COMMIT;
```

## Concurrency Control

Concurrency control is fundamental to database systems, ensuring that multiple transactions execute correctly and securely when they access shared resources. This class covers lock-based protocols, deadlock management, timestamp protocols, multi-version concurrency, and concurrency control in index structures.

### 1. Lock-Based Protocols

Locks control concurrent access to data items, ensuring correct execution across multiple transactions. Locks can be in two modes:

1. **Exclusive (X) Lock**: Allows read and write access. Only one transaction can hold an X-lock at a time.
2. **Shared (S) Lock**: Permits read-only access. Multiple transactions can share this lock simultaneously.

Lock requests are managed by a concurrency-control manager, granting permissions based on a **lock-compatibility matrix**. For example, if one transaction holds an S-lock, other transactions can still read, but if an X-lock is held, all others must wait.

### 2. Locking Protocols and Serializability

A **locking protocol** specifies rules for transactions in requesting and releasing locks, enforcing serializable schedules by limiting concurrent actions.

#### *Schedule Example with Locks*

In the schedule below, transaction T1 reads and writes to X while T2 waits due to incompatible locks:

- T1: lock-X(X), write(X), unlock(X)

- T2: waits for X to be unlocked before proceeding.

This protocol prevents conflicts by enforcing serializability through lock-based restrictions.

### 3. Deadlock and Starvation

A **deadlock** occurs when transactions wait indefinitely due to cyclic dependencies in resource locking.

- **Example:** Transaction T3 locks A and waits for B, while T4 locks B and waits for A.

Deadlock resolution methods:

1. **Timeouts:** Abort transactions after a wait limit.
2. **Deadlock Detection:** Use a **wait-for graph** to detect cycles, triggering rollback.

**Starvation** can also occur if a transaction repeatedly waits, often resolved by managing lock priorities or ensuring fair scheduling.

### 4. Two-Phase Locking Protocol (2PL)

The **two-phase locking protocol** ensures serializability by dividing transactions into two phases:

1. **Growing Phase:** Transactions acquire locks without releasing any.
2. **Shrinking Phase:** Locks are released without acquiring new ones.

This protocol guarantees that transactions execute in a serializable order based on the order of their final lock acquisitions, known as **lock points**.

### 5. Lock Conversions and Automatic Locking

**Lock Conversion** allows changing lock types within two-phase locking:

- **Upgrade** from S-lock to X-lock in the growing phase.
- **Downgrade** from X-lock to S-lock in the shrinking phase.

**Automatic Locking** processes standard read/write commands without explicit lock requests. For instance, a read(D) command automatically acquires an S-lock if compatible or waits if incompatible.

### 6. Implementation of Locking

A **lock manager** tracks granted locks and pending requests in a **lock table**:

- **Granted Locks:** Show current lock status.

- **Pending Requests:** Queue of transactions waiting for specific locks.

**Example Lock Table:**

**Data Item Lock Type Granted To Waiting Transactions**

X	X	T1	T2, T3
Y	S	T2, T4	T5

## 7. Graph-Based Protocols

Graph-based protocols enforce concurrency by arranging data items in a directed acyclic graph (DAG). Transactions can only access data in a specified order, reducing the risk of deadlocks and cycles. The **tree protocol** is a common graph-based protocol that ensures serializability and deadlock freedom by requiring transactions to lock data items in a tree hierarchy, starting from the root.

## 8. Deadlock Handling and Prevention

**Deadlock Prevention** ensures that the system does not enter a deadlock state. Techniques include:

- **Wait-die scheme:** Older transactions wait for younger ones, but younger transactions rollback for older ones.
- **Wound-wait scheme:** Older transactions roll back younger ones instead of waiting.

**Deadlock Detection** uses a wait-for graph where a cycle indicates a deadlock, prompting rollback of the least costly transaction.

## 9. Multiple Granularity Locking

Multiple granularity locking allows locks at various data levels, such as tables, rows, or fields. It's organized in a hierarchy from larger (coarse-grained) to smaller (fine-grained) items, providing higher concurrency while balancing overhead. Transactions follow a set of **intention locks** to manage their access levels:

Lock Type	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	No	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	No	No	No	No	No

**Lock Type IS IX S SIX X**

X	No	No	No	No	No
---	----	----	----	----	----

**10. Phantom Phenomenon**

The **phantom phenomenon** occurs when a transaction reads data that changes mid-operation, causing non-serializable schedules. For instance, a query counting employees in a department might miss new additions due to concurrent inserts.

To handle phantoms, transactions can lock related index entries or use **predicate locks** on queried conditions, ensuring a stable view.

**11. Timestamp-Based Concurrency Control**

In **timestamp-based protocols**, transactions are assigned timestamps upon entry, determining the order of their actions:

- **Read/Write Rules:** Transactions read or write only if their timestamps align with the current record status.
- **Conflict Resolution:** Older timestamps wait while newer ones rollback if conflicts arise.

For example, if transaction T1 has a timestamp of 5 and T2 has a timestamp of 6, T1 must complete all operations on a data item before T2 can proceed.

**12. Multiversion Concurrency Control (MVCC)**

**Multiversion Concurrency Control (MVCC)** maintains several versions of data items to increase concurrency:

- Each write generates a new version, while reads retrieve the appropriate version based on transaction timestamps.
- **Example:** If T1 reads version V1 of X while T2 creates V2 of X, T1 continues using V1, ensuring non-blocking reads.

**Snapshot Isolation** provides each transaction with a consistent snapshot of data, preventing conflicts. It uses the **first-committer-wins** rule, where a transaction only commits if no concurrent transaction has modified its intended updates.

**13. Weak Levels of Consistency**

Weaker isolation levels balance concurrency and consistency. SQL allows several isolation levels:

1. **Serializable:** Ensures full isolation, avoiding all anomalies.
2. **Repeatable Read:** Reads committed records and repeatable data but doesn't prevent phantom reads.
3. **Read Committed:** Allows reading only committed data.
4. **Read Uncommitted:** Reads uncommitted changes, risking dirty reads.

SQL's default level varies but commonly is **Read Committed** for performance.

#### 14. Concurrency Control in Index Structures

Indexes are accessed frequently, requiring specialized concurrency protocols. Common techniques include:

- **Crabbing Protocol:** Locks nodes while moving down a B+-tree, releasing locks as it progresses.
- **B-Link Tree Protocol:** Improves concurrency by unlocking parent nodes before locking child nodes, handling mid-operation structural changes.

### Recovery System

A robust recovery system is vital for ensuring database consistency and reliability in case of errors, crashes, or failures.

#### 1. Failure Classification

Failures can occur at various levels and are classified as follows:

- **Transaction Failure:** Internal errors that prevent a transaction from completing.
  - **Logical errors:** Transaction issues like constraint violations.
  - **System errors:** Database termination of transactions due to deadlocks or other internal conflicts.
- **System Crash:** Hardware or software failures, such as power outages, causing the system to crash.
  - **Fail-Stop Assumption:** Presumes non-volatile storage remains intact despite crashes.
- **Disk Failure:** Physical damage like a head crash, causing data loss.
  - **Detection:** Disk checksums detect storage failures.

#### 2. Recovery Algorithms

Recovery algorithms ensure database consistency by maintaining atomicity and durability.

**Example:** For a transaction  $T_i$  transferring \$50 from account A to B:

1. Subtract \$50 from A.
2. Add \$50 to B.

If a failure occurs after the first step, the database might enter an inconsistent state. Recovery algorithms:

1. Gather sufficient data during regular operations.
2. Use this data post-failure to restore the database to a consistent state.

### 3. Storage Structure

Storage in databases can be classified into three types:

- **Volatile Storage:** Does not survive crashes (e.g., main memory).
- **Nonvolatile Storage:** Survives system crashes (e.g., disk, SSD).
- **Stable Storage:** Simulated by maintaining copies on distinct nonvolatile media to approximate resilience to all failures.

**Stable Storage Implementation:**

1. **Replication:** Store multiple copies of each block on separate disks, sometimes at remote sites.
2. **Protection Against Failures During Transfers:**
  - Write to one physical block.
  - Upon success, write to a secondary block.
  - Completion requires both copies to succeed.

### 4. Data Access

Data items can reside temporarily in memory as **buffer blocks** or permanently on disk as **physical blocks**.

1. **Input Operation:** Loads a disk block to memory.
2. **Output Operation:** Writes a memory buffer to disk.

Each transaction  $T_i$  uses a private workspace, where read(X) and write(X) operations load and modify local copies before committing the changes to disk.

### 5. Recovery and Atomicity

To ensure atomicity, transactions record changes in **stable storage logs** before modifying the database.

#### *Log-Based Recovery*

A **log** is a sequence of records detailing updates on the database. Key log types:



1.  $\langle T_i \text{ start} \rangle$ : Marks the start of a transaction.
2.  $\langle T_i, X, V1, V2 \rangle$ : Logs the update of data item X, showing both old and new values.
3.  $\langle T_i \text{ commit} \rangle$ : Indicates successful transaction completion.

Two main log-based approaches:

1. **Immediate Modification**: Updates are logged and written to memory before transaction commits.
2. **Deferred Modification**: Changes are logged but only written to disk when the transaction commits.

## 6. Transaction Commit

A transaction **commits** once its log entries are securely saved. Log entries are written before database updates, ensuring durability even if system crashes occur.

**Example:** For transaction  $T_0$ , the log entries might appear as:

1.  $\langle T_0 \text{ start} \rangle$
2.  $\langle T_0, A, 1000, 950 \rangle$
3.  $\langle T_0 \text{ commit} \rangle$

The entry  $\langle T_0 \text{ commit} \rangle$  ensures all updates by  $T_0$  are saved before any subsequent failure.

## 7. Undo and Redo Operations

Recovery algorithms define two main operations:

1. **Undo**: Reverts data items to their original values using the log.
2. **Redo**: Reapplies transaction updates from the log to ensure committed changes.

**Example:**

1. If  $T_1$  changes A from 200 to 150, the **undo** operation resets A to 200.
2. A **redo** operation ensures any incomplete updates are finalized.

## 8. Checkpoints

**Checkpoints** streamline recovery by periodically saving the database state, thus reducing the number of log records to process during recovery.

**Checkpoint Process:**

1. Save all in-memory log records.
2. Save all modified buffers to disk.
3. Write  $\langle \text{checkpoint } L \rangle$ , listing all active transactions.

During recovery, only transactions post-checkpoint require processing, optimizing recovery time.

**Example:** Transactions T<sub>2</sub> and T<sub>3</sub> started after the checkpoint may require redo, while T<sub>4</sub> started before and may be ignored if it was committed.

## 9. Log-Based Recovery Algorithms

Recovery methods operate in two phases:

1. **Redo Phase:** Reapplies updates from committed transactions.
2. **Undo Phase:** Reverts uncommitted changes.

During recovery, if  $\langle T_i \text{ start} \rangle$  exists but not  $\langle T_i \text{ commit} \rangle$ , T<sub>i</sub> undergoes undo operations.

## 10. Log Record Buffering

**Log Buffering** improves performance by temporarily storing logs in memory, committing them in bulk to stable storage. Before data output, all related log records must be written to storage (known as **Write-Ahead Logging** or WAL).

## 11. Database Buffering

Database buffering maintains recently accessed data blocks in memory, employing strategies to decide when to write data to disk.

- **No-Force Policy:** Updated blocks aren't written to disk immediately, reducing I/O.
- **Steal Policy:** Permits uncommitted updates to be written to disk, aiding recovery.

**Write-Ahead Logging (WAL)** requires all relevant log records to be saved before data blocks are written to ensure atomicity and consistency.

## 12. Fuzzy Checkpointing

**Fuzzy Checkpointing** reduces checkpoint interruptions:

1. Pause updates temporarily.
2. Write  $\langle \text{checkpoint } L \rangle$  to log.
3. Note modified blocks and save them to disk, allowing transactions to continue in the meantime.

**Advantage:** Minimal interruption to transaction processing, as checkpoints are handled asynchronously.

### 13. Failure Recovery and Storage Dumping

In severe cases, like loss of nonvolatile storage, a **full dump** (copy of the database) to stable storage enables restoration.

During recovery:

1. Restore from the last dump.
2. Use log entries to reapply committed transactions.

### 14. Remote Backup Systems

Remote backup systems improve database availability by maintaining transaction copies at separate locations.

- **Heartbeat Mechanism:** Backup sites monitor primary sites to detect failures.
- **Transfer of Control:** Upon primary failure, the backup site uses the latest logs for recovery, performing a takeover.

#### Backup Configurations:

- **One-safe:** Commits only at primary, risking backup lag.
- **Two-safe:** Ensures transactions are committed at both sites.
- **Hot-Spare:** Continuously applies logs, enabling quick takeover.

---

### PL/SQL (Procedural Language/Structured Query Language)

PL/SQL (Procedural Language/Structured Query Language) is a powerful procedural extension to SQL, developed by Oracle for use within its relational database management systems (RDBMS). It enables the creation of complex, efficient programs that can execute SQL commands along with procedural logic, making it suitable for enterprise-level database applications. Here's an in-depth look:

#### Key Concepts and Features

1. Block Structure:
  - PL/SQL code is organized into blocks, each containing three main sections:
    - **Declaration:** Variables, constants, cursors, and subprograms are declared here.
    - **Execution:** Contains the SQL statements and procedural code to be executed.
    - **Exception Handling:** Defines how errors and exceptions are managed within the block.
  - PL/SQL blocks are either anonymous (unlabeled, temporary blocks) or named (stored as functions, procedures, or triggers in the database).

## 2. Control Structures:

- Conditional Statements: Use IF-THEN-ELSE statements to execute specific code blocks based on conditions.
- Loops: Supports several looping constructs:
  - LOOP: General-purpose loop, which can be terminated with EXIT.
  - WHILE LOOP: Repeats as long as a specified condition is true.
  - FOR LOOP: Runs a loop a predefined number of times.
- Branching Statements: Includes GOTO, which allows jumping to a specific point in the code for complex logic.

## 3. Cursors:

- Implicit Cursors: Automatically created by Oracle when an SQL query is executed. Suitable for single-row queries.
- Explicit Cursors: Used for handling queries that return multiple rows, allowing row-by-row processing.

## 4. Error Handling:

- Exception Handling: PL/SQL has robust exception-handling capabilities with predefined exceptions (like NO\_DATA\_FOUND and TOO\_MANY\_ROWS) and user-defined exceptions, which improve application reliability.
- Built-in Exceptions: Oracle provides several predefined exceptions for common errors, which can be handled within the exception block.

## 5. Stored Procedures and Functions:

- Procedures: Named PL/SQL blocks that perform specific tasks but do not return values. They can take input (IN), output (OUT), or both (IN OUT) parameters.
- Functions: Similar to procedures but always return a single value, making them ideal for calculations and data transformations.
- Stored procedures and functions are reusable, reducing redundancy in code.

## 6. Triggers:

- Database Triggers: PL/SQL code that automatically executes in response to certain events on a table or view, such as INSERT, UPDATE, or DELETE.
- Triggers are useful for enforcing complex business rules, auditing changes, and maintaining data integrity.

## 7. Packages:

- Packages: Group related procedures, functions, cursors, and variables together, improving code organization, security, and reusability.
  - Packages contain two parts:
    - Specification: Declares public elements available to the outside.
    - Body: Contains the actual implementation of the procedures and functions.
8. Collections and Record Types:
- Support data structures like arrays in the form of VARRAYS, nested tables, and associative arrays (index-by tables).
  - Allow grouping multiple related variables into a single composite data type, useful for handling rows of data.
9. Dynamic SQL:
- Dynamic SQL: Allows SQL statements to be constructed and executed at runtime using the EXECUTE IMMEDIATE command. This feature is useful for situations where the SQL structure isn't known until runtime.

#### Advantages of PL/SQL

1. PL/SQL reduces network traffic by allowing multiple SQL statements to be sent to the database server in a single block.
2. Stored procedures and functions hide the implementation details, allowing users to perform operations without direct table access.
3. Blocks, procedures, functions, and packages create modular code, making it easier to read, maintain, and debug.
4. Advanced exception handling helps in managing runtime errors effectively, increasing application robustness.

Examples:

#### **Program: WAP in PL/SQL for adding two numbers.**

```
DECLARE

num1 NUMBER := 10; -- First number

num2 NUMBER := 20; -- Second number

sum NUMBER;      -- Variable to store the sum

BEGIN
```

```

-- Calculate the sum

sum := num1 + num2;

-- Display the result

DBMS_OUTPUT.PUT_LINE('The sum of ' || num1 || ' and ' || num2 || ' is: ' || sum);

END;

/

```

**Output:**

The screenshot shows the OneCompiler IDE interface. The code editor on the left contains the following PL/SQL code:

```

1 DECLARE
2   num1 NUMBER := 10; -- First number
3   num2 NUMBER := 20; -- Second number
4   sum  NUMBER;      -- Variable to store the sum
5 BEGIN
6   -- Calculate the sum
7   sum := num1 + num2;
8
9   -- Display the result
10  DBMS_OUTPUT.PUT_LINE('The sum of ' || num1 || ' and ' || num2 || ' is: ' || sum);
11 END;
12 /
13

```

The output window on the right shows the result of the execution:

```

STDIN
Input for the program (Optional)

Output:
The sum of 10 and 20 is: 30

```

Fig.8: Output screen of Sum of two numbers in PL\SQL

**Program: WAP in PL/SQL for reversing the number. For example the number is 12345 and reverse number will be 54321.**

```

DECLARE

original_num NUMBER := 12345; -- Original number to be reversed

```

```
reversed_num NUMBER := 0;    -- Variable to store the reversed number
remainder  NUMBER;          -- Variable to store the remainder
BEGIN
  -- Display the original number
  DBMS_OUTPUT.PUT_LINE('Original number: ' || original_num);

  -- Reverse the number
  WHILE original_num > 0 LOOP
    remainder := MOD(original_num, 10);    -- Get the last digit
    reversed_num := (reversed_num * 10) + remainder; -- Append the last digit to the reversed
number
    original_num := TRUNC(original_num / 10);    -- Remove the last digit from the original
number
  END LOOP;

  -- Display the reversed number
  DBMS_OUTPUT.PUT_LINE('Reversed number: ' || reversed_num);
END;
/
```

The screenshot shows the OneCompiler interface with a PL/SQL query editor. The code defines variables for the original number (12345), the reversed number, and the remainder. It uses a WHILE loop to reverse the number by taking the last digit and appending it to the reversed number. The output shows the original number as 12345 and the reversed number as 54321.

```

1 DECLARE
2   original_num NUMBER := 12345; -- Original number to be reversed
3   reversed_num NUMBER := 0;    -- Variable to store the reversed number
4   remainder    NUMBER;        -- Variable to store the remainder
5 BEGIN
6   -- Display the original number
7   DBMS_OUTPUT.PUT_LINE('Original number: ' || original_num);
8
9   -- Reverse the number
10  WHILE original_num > 0 LOOP
11    remainder := MOD(original_num, 10); -- Get the Last digit
12    reversed_num := (reversed_num * 10) + remainder; -- Append the Last digit to the rev
13    original_num := TRUNC(original_num / 10); -- Remove the Last digit from the o
14  END LOOP;
15
16  -- Display the reversed number
17  DBMS_OUTPUT.PUT_LINE('Reversed number: ' || reversed_num);
18 END;
19 /
20

```

Output:

```

Original number: 12345
Reversed number: 54321

```

Fig.9: Output screen of reverse number in PL/SQL

### Program: WAP in PL/SQL to find the number is even or odd.

```

DECLARE
  num NUMBER := 7; -- Number to check (you can change this value)
BEGIN
  -- Check if the number is even or odd
  IF MOD(num, 2) = 0 THEN
    DBMS_OUTPUT.PUT_LINE(num || ' is an Even number.');
```

```

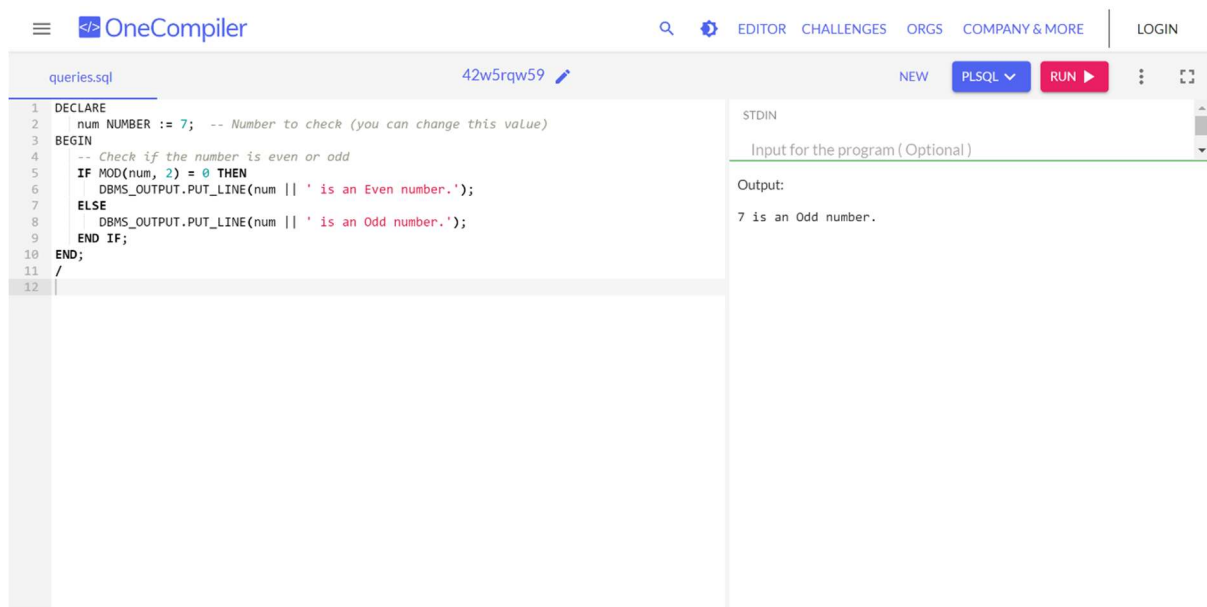
ELSE
  DBMS_OUTPUT.PUT_LINE(num || ' is an Odd number.');
```

```

END IF;
END;
/

```





```

1 DECLARE
2   num NUMBER := 7; -- Number to check (you can change this value)
3 BEGIN
4   -- Check if the number is even or odd
5   IF MOD(num, 2) = 0 THEN
6     DBMS_OUTPUT.PUT_LINE(num || ' is an Even number. ');
7   ELSE
8     DBMS_OUTPUT.PUT_LINE(num || ' is an Odd number. ');
9   END IF;
10 END;
11 /

```

Output:  
7 is an Odd number.

Fig.10: Output screen of even/odd number in PL/SQL

**Program: WAP in PL/SQL to count numbers from 1 to 20.**

```

DECLARE

counter NUMBER := 1; -- Start counter from 1

BEGIN

WHILE counter <= 20 LOOP

    DBMS_OUTPUT.PUT_LINE(counter); -- Display the current number

    counter := counter + 1;    -- Increment the counter by 1

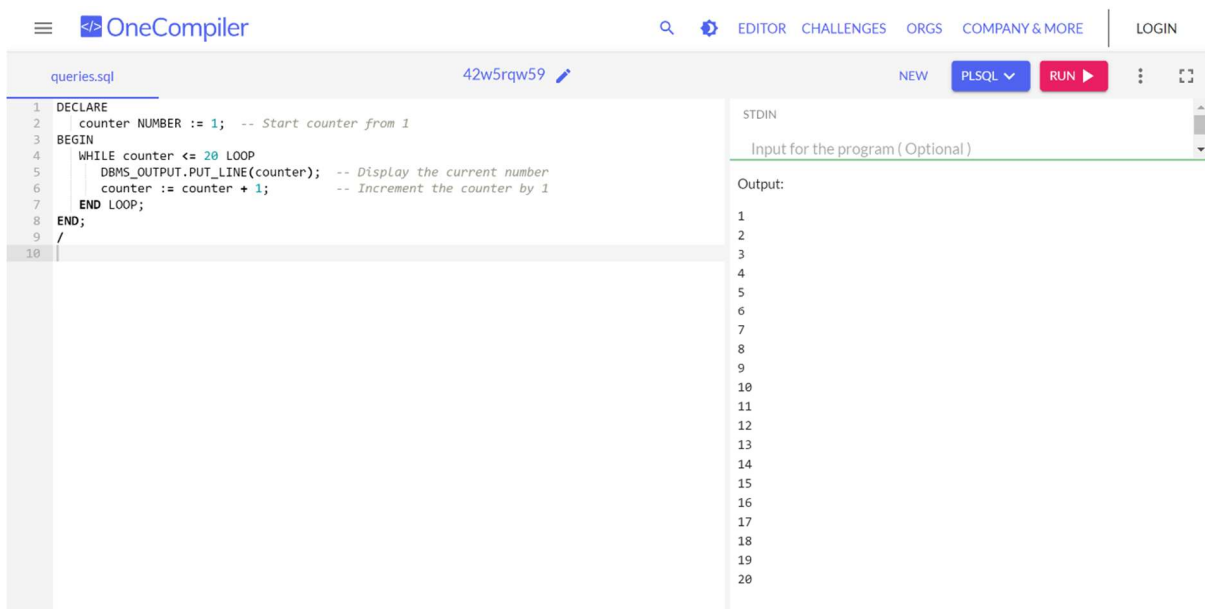
END LOOP;

END;

/

```

**Output**



The screenshot shows the OneCompiler web interface. The editor contains the following PL/SQL code:

```
1 DECLARE
2   counter NUMBER := 1; -- Start counter from 1
3 BEGIN
4   WHILE counter <= 20 LOOP
5     DBMS_OUTPUT.PUT_LINE(counter); -- Display the current number
6     counter := counter + 1; -- Increment the counter by 1
7   END LOOP;
8 END;
```

The output window on the right displays the numbers 1 through 20, one per line.

Fig.11: Output screen of count numbers from 1 to 20 in PL\SQL