Computer Science 275
Database Management Notes

Carter Clifton

Winter 2025

Contents

Lecture 1

Outline

- What is a Database?
- What is a Database Management System (DBMS)?
- Importance of Databases
- Why Do We Need Database Management Systems?
- Types of Databases
- Types of Data Models
- Real-World Applications of Databases

What is a Database?

- A database is an organized collection of data stored electronically to support efficient access, management, and updating.
- Example: A library catalog that stores details about books, authors, and availability.

What is a Database Management System (DBMS)?

- A DBMS is software that provides tools to store, retrieve, and manipulate data efficiently while ensuring data integrity and security.
- Examples: MySQL, Oracle Database, Microsoft SQL Server.

Importance of Databases

- Centralized Data Storage: Avoids data redundancy and ensures consistency.
- Efficient Data Retrieval: Quick access to required data using queries.
- Data Security: Protects sensitive information from unauthorized access.
- Support for Multiple Users: Enables concurrent access without conflicts.

Database Pros:

- Improves data sharing and employees' productivity.
- Removes data redundancy.

- Provides greater security and privacy of data.
- Shows you an integrated view of your business.
- Raises your ability to increase profits as helps you understand better your business operations.
- Ensures consistency of data.
- Provides robust backup and recovery.
- Provides a clean and centralized view of your customers and helps you improve your marketing as well as serve better your customers.

Database Cons:

- Increased costs. The cost of maintaining the DBMS can include advanced hardware training, licensing, regulatory compliance, skilled employees, etc.
- Complex functionality that requires specific and appropriate skills.
- The failure of the database can have a severe negative impact on operations.

Why Do We Need Database Management Systems?

Challenges with File Systems

- Data redundancy and inconsistency.
- Difficulty in accessing and updating data.
- Limited data security and concurrency control.

Feature	**File System**	**DBMS**
Data Redundancy	High	Low
Data Consistency	Difficult to Maintain	Ensured
Query Language	Not Available	SQL Support
Concurrent Access	Limited	Efficient
Security	Minimal	Robust

Advantages of DBMS

- Reduces Data Redundancy: Maintains a single version of data.
- Ensures Data Integrity: Validates data consistency across the system.
- Improves Security: Restricts unauthorized access.
- Provides Query Languages: Enables efficient data manipulation using SQL.
- Scales with Demand: Handles large data volumes and concurrent users effectively.

Types of Databases

Distributed Databases

- Data is stored across multiple locations for redundancy and scalability.
- Example: Google Spanner

Relational Databases

- Data is organized into tables with rows and columns.
- Examples: MySQL, PostgreSQL

NoSQL Database

- Stores unstructured or semi-structured data.
- Examples: Document-based (MongoDB), Key-Value (Redis)

Cloud Database

- Hosted on cloud platforms for easy access and scalability.
- Examples: Amazon RDS, Google BigQuery

What is a Data Model?

- A data model is a conceptual framework that defines how data is structured, organized, and manipulated in a database.
- It serves as a blueprint for designing and implementing databases, helping to standardize how data is stored and accessed.

Key purposes of a data model:

1. Representation: It represents real-world entities and their relationships.
2. Consistency: Ensures data consistency across the database.
3. Guidance: Provides a structure for database developers to design efficient databases.
4. Abstraction: Hides the complexity of data storage while offering a clear interface for users.

Types of Data Models

Hierarchical Data Model

- Description: Represents data in a tree-like structure, where each parent node can have multiple child nodes, but each child node has only one parent.
- Examples: Organizational charts, file systems.
- Advantages:
  - Fast data retrieval for hierarchical relationships.
  - Simple to implement for one-to-many relationships.
- Disadvantages:
  - Lacks flexibility for complex relationships.
  - Difficult to restructure.

Network Data Model

- Description: Represents data using a graph structure where entities are nodes, and relationships are edges. Supports many-to-many relationships.
- Examples: Social networks, supply chain systems.
- Advantages:

  - More flexible than the hierarchical model.
  - Efficient for complex relationships.
- Disadvantages:
  - Complex to implement and manage.
  - Requires advanced query mechanisms.

Relational Data Model

- Description: Organizes data into tables (relations) with rows (tuples) and columns (attributes). Relationships are defined using keys.
- Examples: Employee database, e-commerce systems.
- Advantages:
  - Easy to understand and use.
  - Flexible for querying using SQL.
  - Supports data integrity and normalization.
- Disadvantages:
  - May require significant computational resources for large-scale systems.

Entity-Relationship (ER) Model

- Description: Uses diagrams to visually represent entities, their attributes, and relationships.
- Examples: University database, hospital management systems.
- Advantages:
  - Intuitive for database design.
  - Clearly defines relationships and constraints.
- Disadvantages:
  - Does not directly map to physical storage.

Document Data Model

- Description: Stores data as documents, typically in JSON or XML format. Common in NoSQL databases.
- Examples: MongoDB, CouchDB
- Advantages:
  - Flexible schema for unstructured data.
  - Efficient for hierarchical data storage.
- Disadvantages:
  - Less suited for complex relationships.
  - Limit support for ACID transactions.

Key-Value Data Model

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

- Examples: Redis, DynamoDB
- Advantages:
  - High performance and scalability.
  - Simple and easy to implement.
- Disadvantages:
  - Limited querying capabilities.
  - Not ideal for complex relationships.

Note: The Document Data Model and Key-Value Data Model are both NoSLQ Models.
Choosing the right Data Model

- Relational Data Model: Ideal for structured data with complex relationships.
- NoSQL Models (Document, Key-Value): Best for unstructured or semi-structured data and scalability needs.
- Hierarchical or Network Models: Suitable for legacy systems or well-defined hierarchical relationships.

Real-World Applications of Databases

Banking

- Purpose: Manages accounts, transactions, and customer data.
- Example: ATMs access centralized databases to process transactions.

E-Commerce

- Purpose: Tracks products, orders, and customer interactions.
- Example: Amazon's database stores millions of product details.

Healthcare

- Purpose: Maintains patient records and doctor schedules.
- Example: Electronic Health Record (EHR) systems.

Transportation

- Purpose: Schedules routes and tracks bookings
- Airline reservation systems.

Summary

- What is a Database?
  - Organized collection of electronic data for efficient access and management.
  - Example: Library catalog.
- What is a Database Management System (DBMS)?
  - Software for storing, retrieving, and manipulating data while ensuring integrity and security.
  - Examples: MySQL, Oracle, SQL Server.

- Importance of Databases
  - Centralized storage, efficient retrieval, enhanced security, and support for multiple users.
  - Pros: Data sharing, security, consistency, backup, better business insights.
  - Cons: High cost, complexity, failure has a large impact on operations.
- Why Do We Need DBMS?
  - Solves file system challenges like redundancy, inconsistency, and limited security.
  - Advantages: Reduces redundancy, improves integrity, security, scalability, and query capabilities (SQL).
- Types of Databases
  - Distributed: Data across multiple locations (e.g., Google Spanner).
  - Relational: Data in tables (e.g., MySQL).
  - NoSQL: Unstructured data (e.g., MongoDB).
  - Cloud: Hosted on cloud platforms (e.g., Amazon RDS).
- Types of Data Models
  - Hierarchical: Tree-like structure (e.g., file systems).
  - Network: Graph structure for many-to-many relationships.
  - Relational: Tables with rows and columns (SQL-based).
  - Entity-Relationship (ER): Visual diagrams for design.
  - Document: JSON / XML-based (MongoDB).
  - Key-Value: Pairs of data (Redis).
- Choosing the Right Data Model
  - Relational for structured data.
  - NoSQL for unstructured data.
  - Hierarchical / Network for legacy systems.
- Real-World Applications
  - Banking: Manages transactions and accounts.
  - E-Commerce: Tracks products and orders.
  - Healthcare: Maintains patient records.
  - Transportation: Schedules routes and bookings.

Lecture 2

Outline

- Data Abstraction
- Instances and Schemas
- Database Languages
- Database Design Phases
- Database Users and Administrators

Data Abstraction

- To improve efficiency, database system developers use complex data structures to represent data in the database.
- Since many database-system users are not computer trained, developers hide the complexity from users through several levels of data abstraction
- Abstraction simplifies the interaction with the database by separating the logical aspects of data management from the physical details of data storage.

Levels of Data Abstraction

1. Physical Level (Storage Level)
    - This is the lowest level of abstraction, describing how data is physically stored in the database, such as on disks or memory.
    - Details Included:
        - File structures (e.g., sequential files, B-trees).
        - Storage allocation and indexing methods.
        - Block sizes and compression techniques.
    - Users:
        - Database administrators (DBAs) and system architects.
    - Example: A database file stored on disk using B-trees for indexing.
2. Logical Level (Conceptual Level)
    - This level describes what data is stored in the database and the relationships between the data. It abstracts away the physical storage details.
    - Details Included:

  - Tables, columns, rows, and constraints.
  - Relationships like primary and foreign keys.
  - High-level schema definition.
- Users:
  - Database designers and developers.
- Example: A "Student" table with attributes like StudentID, Name, and Major, along with relationships to other tables.

3. View Level (External Level)
   - This is the highest level of abstraction, providing a user-specific view of the database. It hides the complexity of both the physical and logical levels.
   - Details Included:
     - Customized data views for different users.
     - Simplified interfaces for querying data.
   - Users:
     - End-users and application developers.
   - Example: A university professor sees only the courses they teach, not the entire database.

Instances and Schemas

Aspect	Instance	Schema
Definition	Data stored in the database at a specific point in time (snapshot).	The overall design or structure of the database.
Nature	Dynamic, changes over time as data is added, updated or deleted.	Static, remains relatively constant.
Analogy	Values of variables in a program at a specific time.	Variable declarations and type definitions in a program.
Levels	Not applicable.	- Physical Schema: Design at the physical storage level. - Logical Schema: Logical organization of data. - View Schema: User-specific views (subschemas).
Purpose	Represents the current state of the database.	Provides the blueprint for how data is structured and managed.

Database Languages

A database language has two types:

- A data-definition language (DDL) to specify the database schema and some additional properties of the data.
- A data-manipulation language (DML) to express database queries and updates.

Data-Definition Language

1. Database Schema Specification: Defined using a Data-Definition Language (DDL)

2. Storage and Access Methods: A specialized DDL, called the data storage and definition language, defines the storage structure and access methods, hiding the implementation details from users.

3. Consistency Constraints: DDL allows defining constraints to ensure data integrity, which the database checks during updates.

- Processing DDL statements generates metadata stored in a data dictionary.
- Data Dictionary: A system-managed table that holds "data about data". The data dictionary is accessed by the database system to validate schema, constraints, and authorizations before performing any operations.

Types of Constraints:

- Domain Constraints: Attributes are associated with specific domains (e.g., integers, characters), restricting the values they can take. These are the simplest integrity constraints.
- Referential Integrity: Ensures that relationships between tables are valid (e.g., foreign key values must exist in the referenced table). Violations result in rejecting the operation.
- Authorization: Differentiates user access types, including read, insert, update, and delete permissions, ensuring controlled data access.

SQL:

- SQL provides a rich DDL that allows one to define tables with data types and integrity constraints.
- For instance, the following SQL DDL statement defines the department table:
  create table department
  (dept name char (20),
  building char (15),
  budget numeric (12,2));

Data-Manipulation Language

A data-manipulation language (DML) is a language that enables users to access or manipulate the data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database.
- Insertion of new information into the database.
- Deletion of information from the database.
- Modification of information stored in the database.

There are basically two types of data-manipulation languages:

1. Procedural DMLs require a user to specify what data are needed and how to get those data.
2. Declarative DMLs (also referred to as nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than procedural DMLs.

- A query is a statement requesting the retrieval of information.
- The portion of a DML that involves information retrieval is called a query language.
- The SQL query language nonprocedural.

- A query takes as input several tables (possibly only one) and always returns a single table.
- Example of an SQL query that finds the names of all instructors in the History department:
  select instructor.name
  from instructor
  where instructor.dept_name = 'History';

Database Design

1. Requirements Analysis
   - Understand and document the needs of the users and the application.
   - Activities:
     - Collect requirements through interviews, surveys, or use case analysis.
     - Identify what data needs to be stored and how it will be accessed.
   - Output: A detailed requires specification document.
2. Conceptual Design
   - Create a high-level, abstract representation of the database structure.
   - Activities:
     - Use Entity-Relationship (ER) models or similar tools to define entities, attributes, and relationships.
     - Focus on what data to store, not how it will be stored.
   - Output: A conceptual schema (e.g., ER diagram).
3. Logical Design
   - Convert the conceptual design into a logical structure suitable for a specific database model (e.g., relational model).
   - Activities:
     - Map entities and relationships into tables, columns, and keys.
     - Normalize the database to reduce redundancy and improve consistency.
   - A logical schema (e.g., relational schema).
4. Physical Design
   - Define the physical storage structure and access methods for the database.
   - Activities:
     - Decide on file structures, indexing, and partitioning.
     - Optimize the database for performance based on expected workloads.
   - Output: A physical schema detailing how data will be stored and accessed.

Database Design Phases:

Requirements → Conceptual Design → Logical Design → Physical Design

Database Users and Administrators

- Database users interact with the database system to retrieve, update, and manage data.
- They can be categorized based on their interaction and access levels:

1. End Users: Individuals who interact with the database for specific tasks.
   Types:
   - Casual Users: Use query interfaces or tools to retrieve data (e.g., managers running reports).
   - Naive / Parametric Users: Use pre-designed applications to perform specific tasks (e.g., bank tellers, retail clerks).
   - Sophisticated Users: Use advanced tools or write queries directly (e.g., analysts).
   - Stand-Alone Users: Maintain personal databases using general-purpose tools (e.g., Microsoft Access).
2. Application Programmers: Developers who write application programs to interact with the database.
   Tasks:
   - Write code for database interactions using APIs or embedded SQL.
   - Create user interfaces for end-users.
3. Specialized Users: Experts who interact with the database using complex queries and techniques.
   Examples:
   - Data scientists analyzing large datasets.
   - Researchers performing specialized computations.

Database Administrators (DBAs)

- A Database Administrator is responsible for managing and maintaining the database system to ensure its reliability, performance, and security.
- Key Responsibilities:
  - Database Design and Implementation:
    * Define schemas and storage structures.
    * Ensure efficient design and normalization.
  - Performance Monitoring and Optimization:
    * Monitor database performance.
    * Optimize queries, indexing, and storage.
  - Backup and Recovery:
    * Create and manage backup strategies.
    * Ensure data recovery in case of system failure.
  - Security and Authorization:
    * Define user roles and permissions.
    * Protect data from unauthorized access.
  - Maintenance and Updates:
    * Apply updates and patches to the database system.
    * Address issues and ensure database availability.
  - Data Integrity and Consistency:
    * Implement constraints and validation rules.
    * Ensure data quality and correctness.

Summary

- Data Abstraction
  - Simplifies database interactions by complexity through abstraction levels:
    - Physical Level: How data is stored (e.g., B-trees, storage allocation).
    - Logical Level: What data is stored and relationships (e.g., tables, keys).
    - View Level: User-specific views hiding storage and logical details.
- Instances and Schemas
  - Instance: Data in the database at a specific time (dynamic).
  - Schema: The overall database design or structure (static).
    - Levels: Physical, Logical, View Schemas.
- Database Languages
  - Data-Definition Language (DDL): Defines schemas, storage, and constraints.
  - Data-Manipulation Language (DML): Accesses and manipulates data.
    - Declarative DML: Specifies what data is needed (e.g., SQL).
    - Procedural DML: Specifies what data and how to retrieve it.
- Database Design Phases
  - Requirements Analysis: Understand user needs.
  - Conceptual Design: High-level schema using ER models.
  - Logical Design: Map to tables, normalize data.
  - Physical Design: Define storage, indexing, and optimization.
- Database Users and Administrators
  - Users:
    - Casual, Naive, Sophisticated, and Stand-Alone users.
    - Application Programmers and Specialized Users.
  - Database Administrators (DBAs):
    - Manage design, performance, security, backups, and data integrity.

Lecture 3

Outline

- Overview of SQL
- History of SQL
- Key Features of SQL
- Basic SQL Commands
- SQL Aggregate Functions
- SQL Queries with Clauses

Overview of SQL (Structured Query Languages)

- SQL is a standardized programming language specifically designed for managing, and manipulating relational databases.
- It allows users to create, retrieve, update, and delete data in databases.
- SQL is essential for interacting with relational database management systems (RDBMS) like MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

History of SQL

Early Development (1970s)

- 1970: Dr. Edgar F. Codd, a researcher at IBM, proposed the relational model in his seminal paper, "A relational Model of Data for Large Shared Data Banks".
  - This model introduced the idea of organizing data into tables (relations) with rows and columns.
- 1974: Donald D. Chamberlin and Raymond F. Boyce at IBM developed a language called SEQUEL (Structured English Query Language) to interact with relational databases based on Codd's model.
  - SEQUEL was later renamed to SQL due to trademark issues.

Modern Developments (2000s and Beyond)

- SQL:2003: Added support for XML data.
- SQL:2008: Introduced advanced features like the MERGE statement.
- SQL:2011: Focused on temporal databases, adding support for time-based queries.
- SQL:2016: Enhanced JSON support to accommodate modern web and application development needs.
- SQL:2019: Introduced new features for working with big data (e.g., Polymorphic Table Functions).

Key Features of SQL

1. Declarative Language: Users specify what data they need, not how to retrieve it.
2. Supports wide range of commands:
   - DDL (Data Definition Language): For defining and modifying database schemas (e.g., CREATE, ALTER, DROP).
   - DML (Data Manipulation Language): For querying and modifying data (e.g., SELECT, INSERT, UPDATE, DELETE).
   - DCL (Data Control Language): For controlling access to the database (e.g., GRANT, REVOKE).
   - TCL (Transaction Control Language): For managing transactions (e.g., COMMIT, ROLLBACK).
3. Platform Independence: SQL is supported by almost all relational database systems.
4. Data Integrity: Enforce rules and constraints to ensure data consistency.
5. Scalability: Efficiently handles large datasets.

Basic SQL Commands

SELECT Statement

- Purpose: Retrieve data from one or more tables.
- Syntax:
  SELECT column1, column2, ...
  FROM table_name
  WHERE condition;
- Example: Retrieve all columns from the employee's table
  SELECT * FROM employees;

employee_id	first_name	last_name	department	salary
1234	Martha	Smith	Staff	40000
9876	Adam	Smith	Admin	50000
3456	John	White	HR	60000

- Example: Retrieve specific columns
  SELECT first_name, last_name FROM employees;

first_name	last_name
Martha	Smith
Adam	Smith
John	White

- Example: Retrieve employees in the HR department
  SELECT * FROM employees WHERE department = 'HR';

employee_id	first_name	last_name	department	salary
3456	John	White	HR	60000

INSERT Statement

- Purpose: Add new rows to a table.

- Syntax:
  INTO INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
- Example: Insert a new employee
  INSERT INTO employees (employee_id,
  first_name, last_name, department, salary)
  VALUES (6546, 'John', 'Doe', 'HR', 60000);

employee_id	first_name	last_name	department	salary
1234	Martha	Smith	Staff	40000
9876	Adam	Smith	Admin	50000
3456	John	White	HR	60000
6546	John	Doe	HR	60000

UPDATE Statement

- Purpose: Modify existing data in a table.
- Syntax:
  UPDATE table_name SET column1 = value1,
  column2 = value2, ... WHERE condition;
- Example: Update the salary of an employee
  UPDATE employees SET salary = 65000
  WHERE first_name = 'John' AND
  last_name = 'Doe';

employee_id	first_name	last_name	department	salary
1234	Martha	Smith	Staff	40000
9876	Adam	Smith	Admin	50000
3456	John	White	HR	60000
6546	John	Doe	HR	65000

DELETE Statement

- Purpose: Remove rows from a table.
- Syntax:
  DELETE FROM table_name
  WHERE condition;
- Example: Delete an employee from the database
  DELETE FROM employees
  WHERE employee_id = 1234;

employee_id	first_name	last_name	department	salary
9876	Adam	Smith	Admin	50000
3456	John	White	HR	60000
6546	John	Doe	HR	65000

SQL Aggregate Functions

- An aggregate function is a function that performs a calculation on a set of values, and returns a single value.
- Aggregate functions are often used with the GROUP BY clause of the SELECT statement.
- The most commonly used SQL aggregate functions are:

  - MIN() - Returns the smallest value within the selected column.
  - MAX() - Returns the largest value within the selected column.
  - COUNT() - Returns the number of rows in a set.
  - SUM() - Returns the total sum of a numerical column.
  - AVG() - Returns the average value of a numerical column.
- Aggregate functions ignore null values (except for COUNT()).

SQL Queries with Clauses

WHERE Clause

- Purpose: Filter rows based on conditions.
- Example: Show that employees with a salary greater than 50000
  SELECT * FROM employees WHERE salary >50000;

employee_id	first_name	last_name	department	salary
3456	John	White	HR	60000
6546	John	Doe	HR	65000

GROUP BY Clause

- Purpose: Group rows that have the same values in specified columns.
- Example: Show the total salary by department:
  SELECT department, SUM(salary) AS total_salary FROM employees GROUP BY department;
  Input:

employee_id	first_name	last_name	department	salary
9876	Adam	Smith	Admin	50000
3456	John	White	HR	60000
6546	John	Doe	HR	65000

  Output:

department	total_salary
Admin	50000
HR	125000

HAVING Cause

- Purpose: Filter grouped rows based on aggregate conditions.
- Example: Show the departments with total salary greater than 100000:
  SELECT department, SUM(salary) AS total_salary
  FROM employees
  GROUP BY department
  HAVING total_salary >100000;

department	total_salary
HR	125000

ORDER BY Clause

- Purpose: Sort the result set in ascending or descending order.

- Example: Sort employees by salary (descending)
  SELECT * FROM employees ORDER BY salary DESC;

employee_id	first_name	last_name	department	salary
6546	John	Doe	HR	65000
3456	John	White	HR	60000
9876	Adam	Smith	Admin	50000

Summary

- Overview of SQL
  - SQL (Structured Query Language) is a standardized language for managing relational databases, supporting creation, retrieval, updating, and deletion of data. It is widely used with systems like MySQL, PostgreSQL, and Oracle.
- History of SQL
  - 1970: Dr. Edgar F. Codd introduced the relational model.
  - 1974: SEQUEL was developed at IBM, later renamed SQL.
  - Modern advancements include support for XML (SQL:2003), temporal databases (SQL:2011), JSON (SQL:2016), and big data (SQL:2019).
- Key Features of SQL
  - Declarative language: Focuses on what data to retrieve, not how.
  - Supports DDL (CREATE, ALTER), DML (SELECT, INSERT), DCL (GRANT, REVOKE), and TCL (COMMIT, ROLLBACK).
  - Platform-independent, enforces data integrity, and scalable for large datasets.
- Basic SQL Commands
  - SELECT: Retrieves data from tables.
    - SELECT * FROM employees; — Gets all columns from 'employees'.
  - INSERT: Adds new rows.
    - INSERT INTO employees VALUES (6546, 'John', 'Doe', 'HR', 60000);
  - UPDATE: Modifies data.
    - UPDATE employees SET salary = 65000 WHERE first_name = 'John';
  - DELETE: Removes rows.
    - DELETE FROM employees WHERE employee_id = 1234;
- SQL Aggregate Functions
  - Perform calculations on a set of values. Common functions:
    - MIN(), MAX(), COUNT(), SUM(), AVG().
- SQL Queries with Clauses
  - WHERE: Filters rows.
    - SELECT * FROM employees WHERE salary >50000;
  - GROUP BY: Groups rows by column values.
    - SELECT department, SUM(salary) as total_salary FROM employees GROUP BY department;

- HAVING: Filters grouped rows.
  - SELECT department FROM employees GROUP BY department HAVING SUM(salary) >100000;
- ORDER BY: Sorts results.
  - SELECT * FROM employees ORDER BY salary DESC;

Lecture 4

Outline

- SQL Logical Operator
- SQL NULL Values
- SQL CREATE / DROP DATABASE Statement
- SQL CREATE / DROP / ALTER TABLE Statement
- SQL Data Types
- SQL Constraints

SQL Logical Operator

- The following SQL statement selects all customers from Spain that starts with a "G" or an "R":
  SELECT * FROM Customers
  WHERE Country = 'Spain' AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');
- Select only the customers that are NOT from Spain:
  SELECT * FROM Customers
  WHERE NOT Country = 'Spain';
- Select customers that do not start with the letter 'A':
  SELECT * FROM Customers
  WHERE CustomerName NOT LIKE 'A%';
- Select customers with a customerID not between 10 and 60:
  SELECT * FROM Customers
  WHERE CustomerID NOT BETWEEN 10 AND 60;
- Select customers that are not from Paris or London
  SELECT * FROM Customers
  WHERE City NOT IN ('Paris', 'London');

SQL NULL Values

What is a NULL Value?

- A field with a NULL value is a field with no value.
- If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.
- Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

SQL CREATE / DROP DATABASE Statement

- The CREATE DATABASE statement is used to create a new SQL database.
- Syntax:
  CREATE DATABASE database_name;
- The DROP DATABASE statement is used to drop an existing SQL database.
- Syntax:
  DROP DATABASE database_name;

SQL CREATE TABLE Statement

- The CREATE TABLE statement is used to create a new table in a database.
- Syntax:
  CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  column3 datatype,
  ...
  );
- Example:
  CREATE TABLE Persons (
  PersonID int,
  LastName varchar(255),
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255)
  );

SQL DROP TABLE Statement

- The DROP TABLE statement is used to drop an existing table in a database.
- Syntax:
  DROP TABLE table_name;
- The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.
- Syntax:
  TRUNCATE TABLE table_name;

SQL ALTER TABLE Statement

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.
- Example: The following SQL adds an "Email" column to the "Customers" table:
  ALTER TABLE Customers
  ADD Email varchar(255);

- Example: The following SQL deletes the "Email" column from the "Customer's" table:
  ALTER TABLE Customers
  DROP COLUMN Email;
- To rename a column in a table, use the following syntax:
  ALTER TABLE table_name
  RENAME COLUMN old_name to new_name;
- To change the data type of a column in a table, use the following syntax:
  ALTER TABLE table_name
  MODIFY COLUMN column_name datatype;

SQL Data Types

- The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on.
- An SQL developer must decide what type of data that will be stored inside each column when creating a table.
- Data types might have different names in different databases (e.g., MySQL, SQL Server, and MS Access). And even if the name is the same, the size and other details may be different! Always check the documentation.
- This course will use MySQL data types.

Data Type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1.
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The size parameter specifies the maximum string length in characters - can be from 0 to 65535.
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1.
BIT(size)	A bit-value type (numerical). The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1.
BOOL	Zero is considered as false, nonzero values are considered as true.
INT(size)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display with (which is 255).
INTEGER(size)	Equal to INT(size).
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'.
DATETIME(fsp)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time.
TIMESTAMP(fsp)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition.
TIME(fsp)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'

- fsp stands for Fractional Seconds Precision. It ranges from 0 to 6.
- 0 means: No fractional seconds are stored (e.g., 2025-01-13 10:30:45).
- 6 means: Stores up to six digits for fractional seconds (e.g., 2025-01-13 10:30:45.123456).

SQL Constraints

- SQL constraints are used to specify rules for data in a table.
- Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.
- Constraints can be column level or table level.
- Syntax:
  CREATE TABLE table_name (
      column1 datatype constraint,
      column2 datatype constraint,
      column3 datatype constraint,
      ...
  );
- The following constraints are commonly used in SQL:
  - NOT NULL — Ensures that a column cannot have a NULL value.
  - UNIQUE — Ensures that all values in a column are different.
  - PRIMARY KEY — A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.
  - FOREIGN KEY — Prevents actions that would destroy links between tables.
  - CHECK — Ensures that the values in a column satisfies a specific condition.
  - DEFAULT — Sets a default value for a column if no value is specified.
  - CREATE INDEX — Used to create and retrieve data from the database very quickly.

SQL Constraint Examples

- SQL NOT NULL on CREATE TABLE
  - The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:
  - Example:
    CREATE TABLE Persons (
        ID int NOT NULL,
        LastName varchar(255) NOT NULL,
        FirstName varchar(255) NOT NULL,
        Age int
    );
  - Example:
    ALTER TABLE Persons
    ALTER COLUMN Age int NOT NULL;
- SQL UNIQUE Constraint
  - The UNIQUE constraint ensures that all values in a column are different.
  - The following SQL creates a UNIQUE constraint on the "ID" column.
  - Example:
    CREATE TABLE Persons (
        ID int NOT NULL UNIQUE,
        LastName varchar(255) NOT NULL,

FirstName varchar(255),
Age int
);

- Example:
ALTER TABLE Persons
ADD UNIQUE (ID);

- SQL PRIMARY KEY Constraint
  - The PRIMARY KEY constraint uniquely identifies each record in a table.
  - Primary keys must contain UNIQUE values, and cannot contain NULL values.
  - A table can have only ONE primary key; and in the table, this primary key can consist of a single or multiple columns (fields).
  - Example:
CREATE TABLE Persons (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int,
PRIMARY KEY (ID)
);
  - Example:
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
  - Example:
ALTER TABLE Persons
DROP PRIMARY KEY;

- SQL FOREIGN KEY Constraint
  - The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
  - A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
  - The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

  - In the first table, PersonID is the primary key, in the second table, PersonID is the foreign key.
  - The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:
CREATE TABLE Orders (
OrderID int NOT NULL,
OrderNumber int NOT NULL,
PersonID int,
PRIMARY KEY (OrderID),
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);

- To create a FOREIGN KEY constraint on the "PersonID" column whe the "Orders" table is already created, use the following SQL:
  ALTER TABLE Orders
  ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

  ALTER TABLE Orders
  DROP FOREIGN KEY FK_PersonOrder;

- SQL CHECK Constraint
  - The CHECK constraint is used to limit the value range that can be placed in a column.
    CREATE TABLE Persons (
        ID int NOT NULL,
        LastName varchar(255) NOT NULL,
        FirstName varchar(255),
        Age int,
        CHECK (Age >= 18)
    );

    ALTER TABLE Persons
    ADD CHECK (Age >= 18);

    ALTER TABLE Persons
    DROP CONSTRAINT CHK_PersonAge;

- SQL DEFAULT Constraint
  - The DEFAULT constraint is used to set a default value for a column.
    CREATE TABLE Persons (
        ID int NOT NULL,
        LastName varchar(255) NOT NULL,
        FirstName varchar(255),
        Age int,
        City varchar(255) DEFAULT 'Sandnes'
    );

    ALTER TABLE Persons
    ALTER City SET DEFAULT 'Sandnes';

    ALTER TABLE Persons
    ALTER City DROP DEFAULT;

Summary

- SQL Logical Operators
  - Using AND, OR, NOT, LIKE, BETWEEN, and IN to filter records.
- SQL Null Values
  - NULL represents missing or undefined data, different from zero or blank spaces.
- SQL Database Management
  - CREATE DATABASE and DROP DATABASE statements for database creation and deletion.
- SQL Table Management
  - CREATE TABLE, DROP TABLE, TRUNCATE TABLE, and ALTER TABLE statements for table creation, modification, and deletion.

- SQL Data Types
  - Includes CHAR, VARCHAR, BINARY, BIT, BOOL, INT, INTEGER, DATE, DATETIME, TIMESTAMP, and TIME.
- SQL Constraints
  - Rules for ensuring data integrity, including:
    * NOT NULL: Prevents NULL values.
    * UNIQUE: Ensures column values are unique.
    * PRIMARY KEY: Combines NOT NULL and UNIQUE for row identification.
    * FOREIGN KEY: Maintains relationships between tables.
    * CHECK: Enforces a specific conditions.
    * DEFAULT: Sets a default column value.
    * CREATE INDEX: Speeds up data retrieval.

Lecture 5

Outline

- SQL Joins
  - (INNER) JOIN
  - LEFT (OUTER) JOIN
  - RIGHT (OUTER) JOIN
  - FULL (OUTER) JOIN
  - SQL Self JOIN
- SQL UNION

SQL Joins

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- Consider the tables:

OrderID	CustomerID	Order Date
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

CustomerID	CustomerName	ContactName	Country
1	Alfred Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados	Ana Trujillo	Mexico
3	Antonio Moreno Taqueria	Antonio Moreno	Mexico

- We can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

OrderID	CustomerID	OrderDate
10308	Ana Trujillo Emparedados	9 / 18 / 1996
10365	Antonio Moreno Taqueria	11 / 27 / 1996
10383	Around the Horn	12 / 16 / 1996
10355	Around the Horn	11 / 15 / 1996
10278	Berglunds Snabbkop	8 / 12 / 1996

Different Types of SQL JOINs

- Here are the different types of JOINs in SQL:
  - (INNER) JOIN: Returns records that have matching values in both tables.
  - LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table.
  - RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table.
  - FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table.

SQL LEFT JOIN

- The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2).
- The result is 0 records from the right side, if there is no match.

  SELECT Customers.CustomerName, Orders.OrderID
  FROM Customers LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
  ORDER BY Customers.CustomerName;

Note: These tables do not include every entry

Table 1:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfred Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo	Ana Trujillo	Avda. de la Constitucion 2222	Mexico D.F.	05021	Mexico
3	Antonio Moreno Taqueria	Antonio Moreno	Mataderos 2312	Mexico D.F.	05023	Mexico

Table 2:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

Result:

CustomerName	OrderID
Alfred Futterkiste	
Ana Trujillo	10308
Antonio Moreno Taqueria	10365
Around the Horn	10365
Around the Horn	10383
Berglunds Snabbkop	10278
Berglunds Snabbkop	10280

SQL RIGHT JOIN

- The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1).
- The result is 0 records from the left side, if there is no match.

  SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
  FROM Orders RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
  ORDER BY Orders.OrderID;

Note: These tables do not include every entry

Table 1:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

Table 2:

EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmployeeD1.pic
2	Fuller	Andrew	2/19/1952	EmployeeD2.pic
3	Leverling	Janet	8/30/1963	EmployeeD3.pic

Result:

OrderID	LastName	FirstName
	West	Adam
10248	Buchanan	Steven
10249	Suyama	Michael
10250	Peacock	Margaret
10251	Leverling	Janet
10252	Peacock	Margaret
10253	Leverling	Janet
10254	Buchanan	Steven

SQL FULL JOIN

- The FULL OUTER JOIN keyword returns all records when there is a match in the left (table1) or right (table2) table records.
- FULL OUTER JOIN and FULL JOIN are the same.

  SELECT Customers.CustomerName, Orders.OrderID
  FROM Customers FULL OUTER JOIN Orders ON Customers.CustomerID = Orders.CustomerID
  ORDER BY Customers.CustomerName;

Note: These tables do not include every entry

Table 1:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfred Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo	Ana Trujillo	Avda. de la Constitucion 2222	Mexico D.F.	05021	Mexico
3	Antonio Moreno Taqueria	Antonio Moreno	Mataderos 2312	Mexico D.F.	05023	Mexico

Table 2:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

Result:

CustomerName	OrderID
Null	10309
Null	10310
Alfred Futterkiste	*Null*
Ana Trujillo	10308
Antonio Moreno Taqueria	*Null*

SQL Self JOIN

- A self join is a regular join, but the table is joined with itself.
- Self Join Syntax:

  SELECT column_name(s)
  FROM table1 T1, table1 T2
  WHERE condition;

- We have an employees table that stores information about employees and their managers.
- Each employee has a manager_id that refers to another employee in the same table (their manager).
- The following query retrieves each employee along with their manager's name using a self join:

  SELECT e1.name AS employee_name, e2.name AS manager_name
  FROM employees e1, employees e2
  WHERE e1.manager_id = e2.employee_id;

Table:

employee_id	name	manager_id
1	Alice	*Null*
2	Bob	1
3	Charlie	1
4	Diana	2
5	Edward	2

Result:

employee_name	manager_name
Alice	*Null*
Bob	Alice
Charlie	Alice
Diana	Bob
Edward	Bob

SQL Union

- The UNION operator is used to combine the result-set of two or more SELECT statements.
  - Every SELECT statement within UNION must have the same number of columns.
  - The columns must also have similar data types.

  - The columns in every SELECT statement must also be in the same order.

- UNION Syntax:

  SELECT column_name(s) FROM table1
  UNION
  SELECT column_name(s) FROM table2;

- The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

  SELECT City FROM Customers
  UNION
  SELECT City FROM Suppliers
  ORDER BY City;

- Note: If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only distinct values. Use UNION ALL to also select duplicate values!

Result:

City
Aachen
Albuquerque
Anchorage
Ann Arbor
Annecy
Arhus
Barcelona

Summary

- SQL Joins
  - Joins are used to combine rows from two or more tables based on a related column.
    * INNER JOIN: Returns records with matching values in both tables.

      SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
      FROM Orders
      INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
    * LEFT JOIN (OUTER JOIN): Returns all records from the left table and matching records from the right table; unmatched records from the right table are NULL.

      SELECT Customers.CustomerName, Orders.OrderID
      FROM Customers
      LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
    * RIGHT JOIN (OUTER JOIN): Returns all records from the right table and matching records from the left table; unmatched records from the left table are NULL.

      SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
      FROM Orders
      RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID;
    * FULL JOIN (FULL OUTER JOIN): Returns all records from both tables, with NULLs in place for non-matching records.

      SELECT Customers.CustomerName, Orders.OrderID
      FROM Customers
      FULL OUTER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

    * Self JOIN: A table is joined with itself, commonly used for hierarchical relationships.

      SELECT e1.name AS employee_name, e2.name AS manager_name
      FROM employees e1, employees e2
      WHERE e1.manager_id = e2.employee_id;

- SQL Union
  - The UNION operator combines result sets of two or more SELECT statements, ensuring unique values unless UNION ALL is used.
    * UNION does not include duplicates.

      SELECT City FROM Customers
      UNION
      SELECT City FROM Suppliers;
    * UNION ALL includes duplicates.

      SELECT CITY FROM Customers
      UNION ALL
      SELECT City FROM Suppliers;

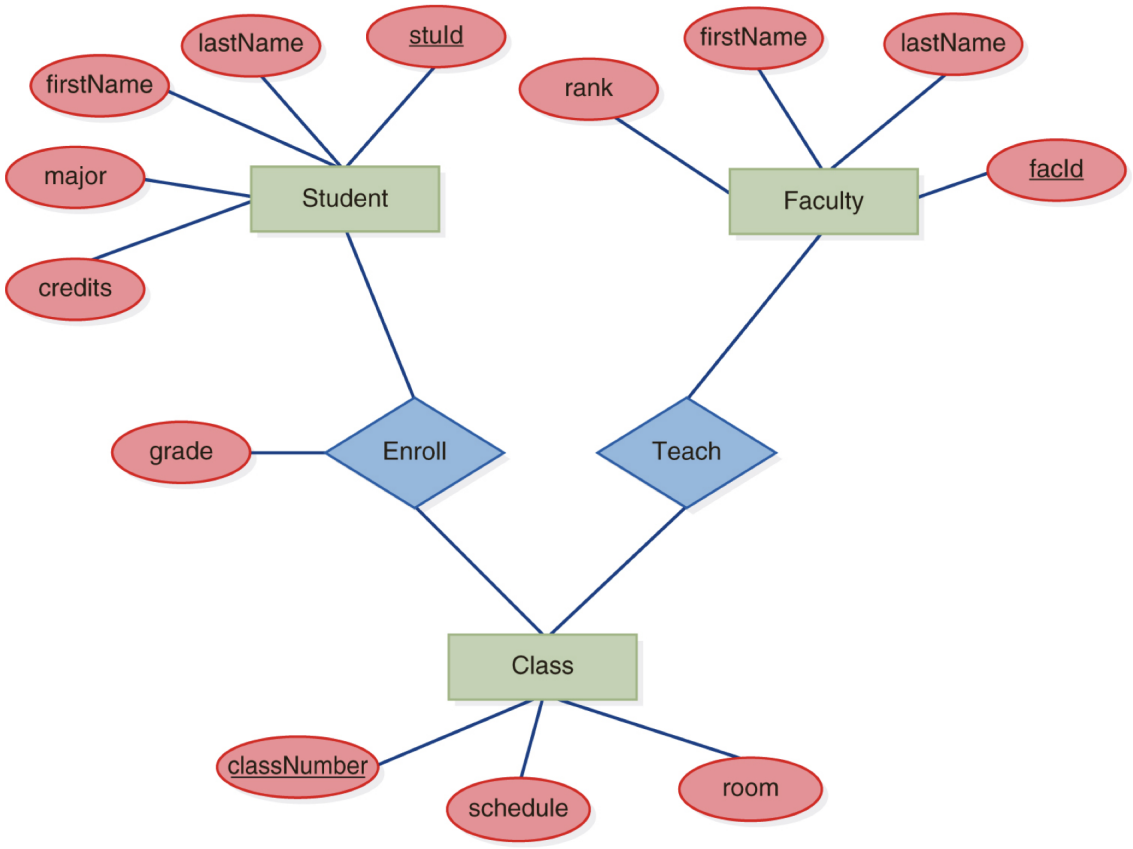
Lectures 6 - 8

Outline

- Introduction to ER Modelling
- Entities and Attributes
- Relationships
- ER Diagrams

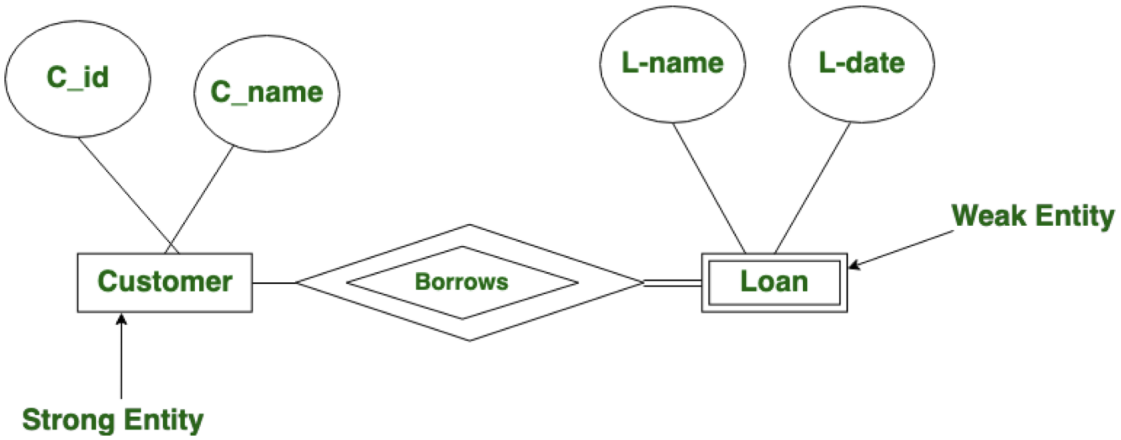
What is Entity-Relationship (ER) Modelling?

- ER modelling is a high-level data modelling technique used to define the structure of a database.
- It represents real-world data and its relationships in a conceptual model.
- Basis for creating relational databases.
- Purpose of ER Diagrams:
  - To visually represent data requirements.
  - Serve as a blueprint for database design.
  - Help stakeholders understand and validate data requirements.
- Core Components of ER Modelling:
  - Entities: Objects or concepts that can have data stored about them.
    - Examples: Students, Courses, Products.
  - Attributes: Properties or details of an entity.
    - Example: Student attributes include Name, StudentID, and Email.
  - Relationships: Associations between entities.
    - Example: "Enrolls" is a relationship between Students and Courses.



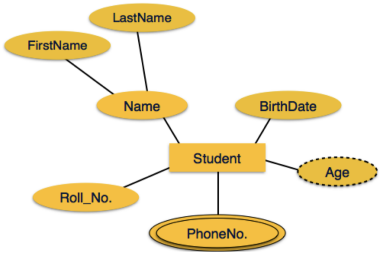
Entities and Attributes

- Types of Entities:
  - Strong Entities
  - Weak Entities

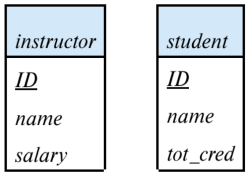


Strong Entity	Weak Entity
Strong entity always has a primary key.	While a weak entity has a partial discriminator key.
Strong entity is not dependent on any other entity.	Weak entity depends on strong entity.
Strong entity is represented by a single rectangle.	Weak entity is represented by a double rectangle.
Two strong entity's relationships is represented by a single diamond.	While the relationship between one strong and one weak entity is represented by a double diamond.
Strong entities either have total participation or partial participation.	A weak entity has a total participation constraint.

- Attribute Representation:
  - Traditional ER Diagrams (Chen's Notation)
    - Attributes are represented as ovals (ellipses) connected to the entity or relationship they describe.

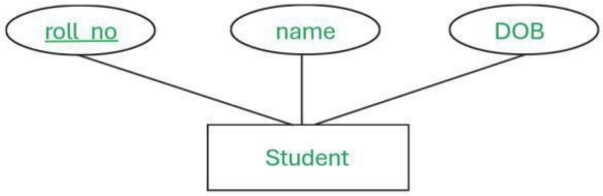


  - Crow's Foot Notation or UML Style
    - Attributes are often listed inside the box representing the entity.
    - Attributes are written as a list below the entity's name, with key attributes (primary keys) often emphasized (e.g., underlined or bold).

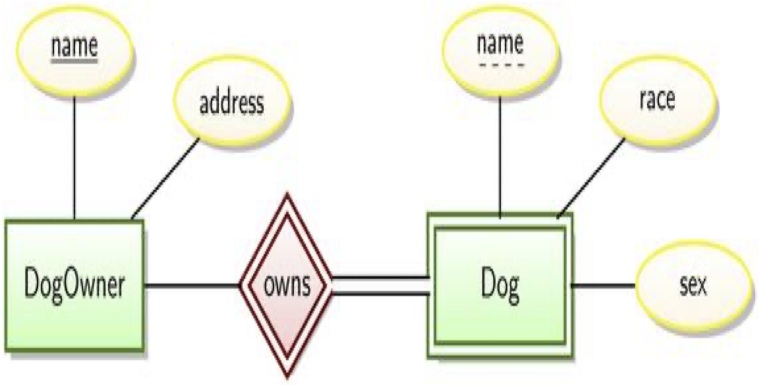


- Types of Attributes:
  - Simple Attributes:
    - Cannot be broken down further.
    - Example Roll_No.
  - Composite Attributes:
    - Can be divided into smaller parts.
    - Example: Address, Name (first and last).
  - Derived Attributes:
    - Values that can be calculated from other attributes.
    - Example: Age (Calculated form Date of Birth).
  - Multi-valued Attributes:
    - Can have multiple values.
    - Example: PhoneNumbers.
  - Key Attribute:
    - Represented as an oval with the attribute name underlined. It represents the primary key.

* Example: Roll_No for a Student Entity.

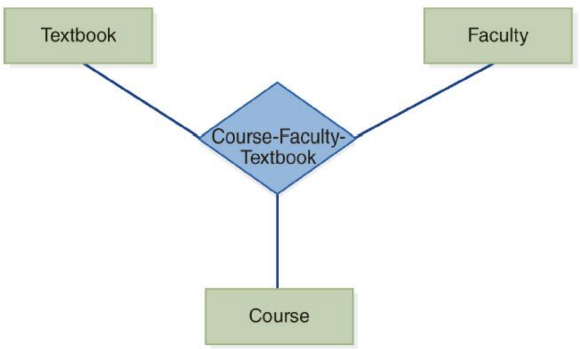


– Weak Key Attribute (Discriminator):
  * Represented as an oval with a dashed underline.
  * Example: Dog name in the Dog entity which is dependant on the Dog Owner.

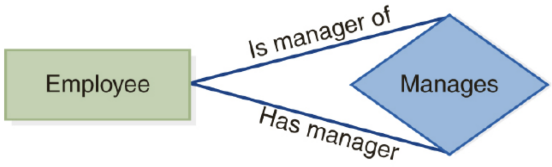


Relationships

- Entities are often linked by associations or relationships, which are connections or interactions among the entity instances.
- For example, a student may be related to a class by being enrolled in that class.
- The name given to a relationship set is chosen by the designer. It is a good practice to choose a verb that captures the meaning of the relationship.
- The degree of a relationship is the number of entity sets that are linked by the relationship.
- A relationship that links two entity sets has degree two and is called binary, one that links three entity sets has degree three and is called ternary, and one that links more than three entity sets is called n-ary.

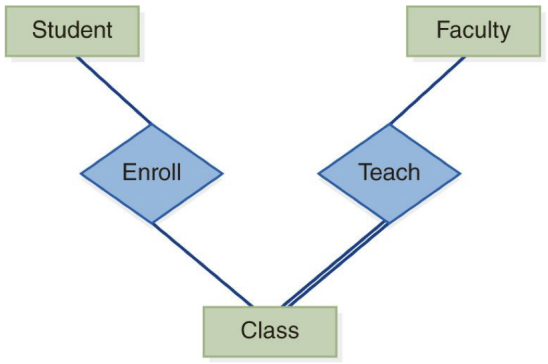


- It is also possible to have a unary relationship, one that links an entity to itself and has degree one.
- The most common type of relationship is binary.
- A relationship in which an entity set is related to itself is called recursive.



- If we had an entity set Employees, we might have a recursive Manages relationship that links each employee to his or her manager, who is also an employee. That set would be defined as:
- Sometimes a relationship set has descriptive attributes that belong to the relationship rather than or any of the entities involved.
- For example, the attribute grade is a descriptive attribute for the Enroll relationship set.
- The attribute grade does not describe the Student entity because each student can have grades for several classes, nor does it describe the Class entity because there are different grades given in a particular class for different students.
- Two types of constraints on relationships are participation constraints and cardinality.
- The cardinality of a binary relationship is the number of entity instances to which another entity instance can map under that relationship at any given time.
- There are four types of binary relationships:
  - One-to-one
    - A relationship R from entity set X to entity set Y is one-to-one if each entity instance in X is associated with at most one entity instance in Y.
    - Example: Every person has one SIN.
  - Many-to-many
    - A relationship R from entity set X to entity set Y is many-to-many if each entity instance in X can be associated with many entity instances in Y and each entity in Y can be associated with many entity instances in X.
    - Example: A student is enrolled in several courses and a course has many students.
  - Many-to-one
    - A relationship R from entity set X to entity set Y is many-to-one if each entity instance in X is associated with at most one entity instance in Y, but each entity instance in Y can be associated with many entity instances in X.
    - Example: Many people can have the same address.
  - One-to-many
    - A relationship R from entity set X to entity set Y is one-to-many if each entity instance in X can be associated with any entity instances in Y, but each entity instance in Y is associated with at most one entity instance in X.
    - Example: A person may have many cars, but a car is owned by one person.
- It is possible that not all members of an entity set participate in a relationship.
- For example, some faculty members may not be teaching this semester.

- If every member of an entity set must participate in a relationship, this is referred to as the total participation of the entity set in the relationship.
- A single line indicates that some members of the entity set may not participate in the relationship, which a situation called partial participation.
- A double line would imply that a person is not a student unless he or she is enrolled in some class.
- This double line means that no faculty member is not teaching any class.



- Single lines or double lines showed participation for relationships.
- Relationship cardinality was shown by using "1" for "one" and "M" (or "m", "N", or "n") for "many" on the appropriate lines.
- An alternative representation that shows both participation and cardinality constraints is the (min, max) notation.

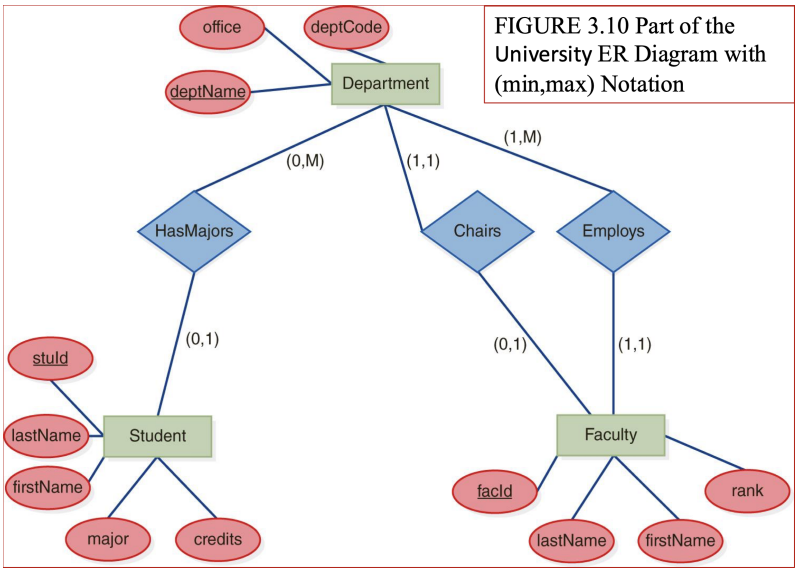


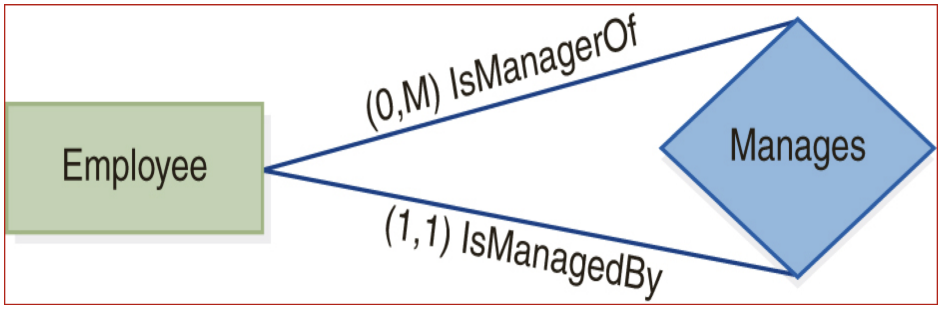
FIGURE 3.10 Part of the University ER Diagram with (min,max) Notation

- Also, observe that the placement of (min, max) is between the entity and the relationship set, not between the entities, and it is the opposite from the placement of the cardinality in the 1:M notation.

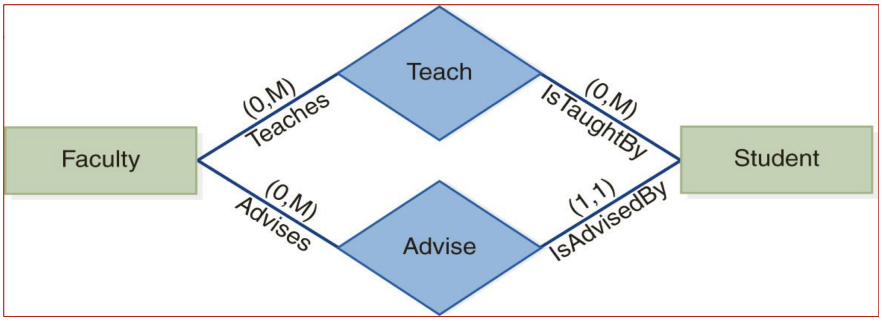
Roles

- In a relationship, each entity has a function called its role in the relationship.
- Usually, it is clear from the context what role an entity plays in a relationship.

- When an entity set is related to itself in a recursive relationship, it is necessary to indicate the roles that members play in the relationship.



- Note that is is a one-to-many relationship in which every employee must have a manager (total participation), but not every employee has to be a manager.
- Another instance where roles names are helpful occurs when the same two entity sets are related in two different ways.
- For example, suppose we wanted to represent two different relationships types between faculty members and students.
  - One type would be the usual Faculty-Student relationship, which we will call Teach in which the Faculty entity has the Teaches role, and the Student entity has the IsTaughtBy role.
  - Another is the Advise relationship, in which the Faculty entity has the Advises role and the Student entity has the IsAdvisedBy role.



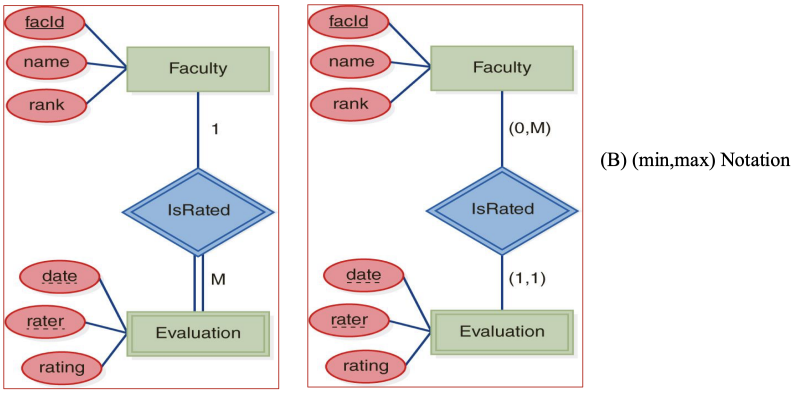
Weak Entites

- A weak entity is an entity that cannot be uniquely identified by its own attributes alone.
- It depends on a strong entity (also called its owner entity) for identification and typically has a partial key.
- A weak entity is always associated with a relationship that links it to its strong entity, which is represented with a double diamond in an ER diagram.
- Characteristics of Weak Entities:
  - Dependence on Strong Entity: A weak entity relies on a strong entity for its primary key.
  - Partial Key: Weak entities have a partial key, which uniquely identifies them within the context of the related strong entity.
  - Existence Dependency: Weak entities cannot exist without the corresponding strong entity.

- A weak entity is depicted in the ER diagram by drawing a double rectangle around the entity and making the relationship diamond a double diamond.

FIGURE 3.12 Weak Entity Set

(A) 1:M and Double Line Notation



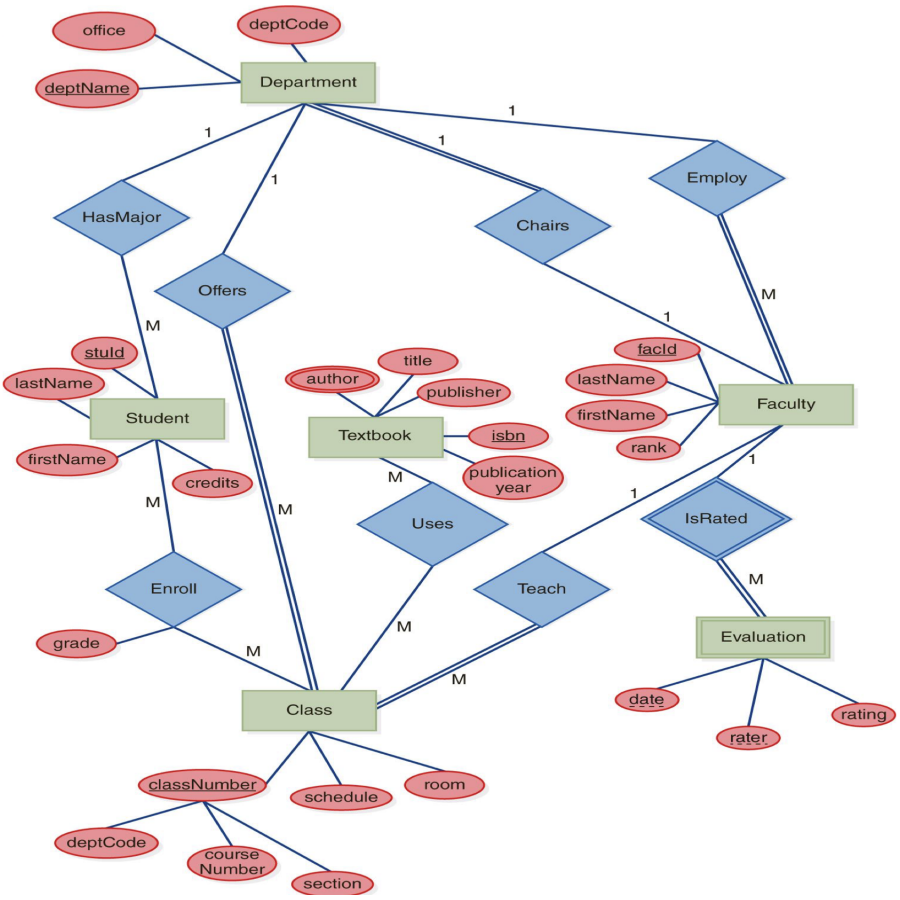
(B) (min,max) Notation

- When an entity is weak, it has no primary key of its own; it is unique only reference to its relationship to its owner.
- However, it often has a partial key, also called a discriminator, that allows us to uniquely identify the weak entities that belong to the same owner.
- The partial key may be a single attribute or a composite.
- The discriminator is indicated by a dashed underline on the ER diagram.
  - For Evaluation, date alone cannot be sufficient, if it is possible for a faculty member to be evaluated by two different people on the same date.
  - Therefore, we will use the combination of date and rater as the discriminator.

Sample ER Diagrams

- To construct an ER diagram, we can begin with a scenario, a brief description of what information about the mini-world the database should contain.
- For a database for a university, let us assume that we want to keep information about students, the classes they are enrolled in, the faculty who teach those classes, and the textbooks used.
- We also want to represent departments that employ the faculty members, the faculty member who chairs each department, and teaching evaluations made of the faculty.
- Identify the entities of the University database:
- Our first task is to identify entities.
- Note that we do not make the enterprise on of the entities, so we will have no entity set called University because this is the entire enterprise. Everything in the diagram represents some facet of the university.
- A good way to identify entities is to look for nouns in the scenario that may be people, places, or events, or concepts.
- For each of these potential entities, we choose attributes that describe them, making assumptions as we record them.
- Ideally, we would consult with users before making these assumptions.
- To determine the attributes, we ask what facts about the entity would be important to keep in the database, and we think about the potential values we might store for those facts.

- We identify relationships.
- Relationships are usually indicated in the scenario by verbs that describe some type of interaction between the entities.
- For example, enroll, teach, employ, uses, made, and chairs, so we try out these relationships, drawing diamonds and connecting the entities to them.
- Then, we decide on cardinality and participation constraints for each relationship.



Summary

- Strong vs. Weak Entities
  - Strong Entity: Has a primary key and is independent.
  - Weak Entity: Lacks a primary key, relies on a strong entity, a nd has a partial key (discriminator).
  - Representation:
    - Strong entities use a single rectangle; weak entities use a double rectangle.
    - Relationships between two strong entities are represented by a single diamond; relationships involving a weak entity use a double diamond.
  - Participation:
    - Strong entities can have total or partial participation.
    - Weak entities always have total participation.
- Attributes in ER Diagrams

  - Simple Attributes: Cannot be broken down (e.g., Roll No).
  - Composite Attributes: Can be divided into subparts (e.g., Address: Street, City).
  - Derived Attributes: Computed from other attributes (e.g., Age from Date of Birth).
  - Multi-Valued Attributes: Can have multiple values (e.g., PhoneNumbers).
  - Key Attributes: Primary keys are underlined.
  - Weak Key Attributes (Discriminator): Represented with a dashed underline.

- Relationships in ER Models
  - Definition: Associations between entity instances (e.g., a student enrolls in a class).
  - Types of Relationships (Based on Cardinality):
    * One-to-One (1:1): Each entity in X is linked to at most one in Y (e.g., A person has one SIN).
    * One-to-Many (1:M): One entity in X is linked to multiple in Y (e.g., A person owns multiple cars).
    * Many-to-One (M:1): Multiple entities in X link to one in Y (e.g., Many people share the same address).
    * Many-to-Many (M:N): Multiple entities in X link to multiple in Y (e.g., Students enroll in multiple courses).
  - Participation Constraints:
    * Total Participation: Every entity must participate in the relationship (double line).
    * Partial Participation: Some entities may not participate (single line).
  - Recursive Relationships: An entity relates to itself (e.g., An employee manages another employee).
  - Roles: When entities have multiple relationships, role names help distinguish them (e.g., Faculty teaches students and advises them).

- Weak Entities and ER Diagrams
  - Dependence on Strong Entity: Must be linked to a strong entity.
  - Existence Dependency: Cannot exist independently.
  - Partial Key (Discriminator): Identifies weak entities within a strong entity.
  - Representation:
    * Double rectangle for weak entities.
    * Double diamond for the relationship.
    * Dashed underline for the discriminator.

- Constructing ER Diagrams
  - Identify Entities: Look for nouns in the problem statement.
  - Choose Attributes: Define key attributes and assumptions.
  - Define Relationships: Identify verbs that describe interactions.
  - Set Constraints: Determine cardinality and participation rules.

Lecture 9

Outline

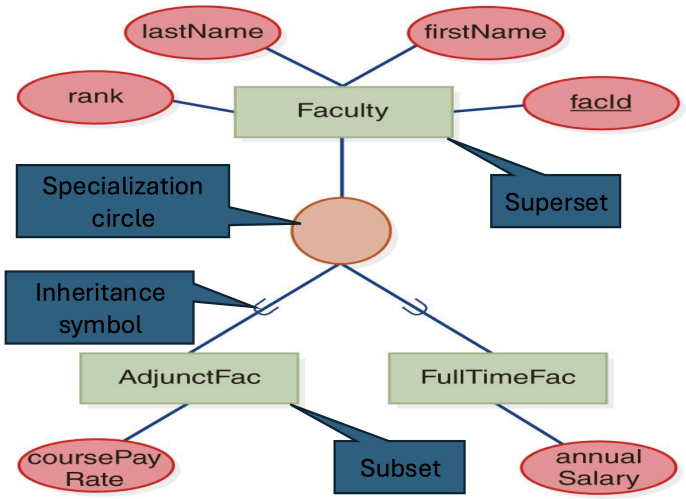
- The Extended Entity-Relationship (EER) Model
- Generalization and Specialization
- Union
- Sample EER Diagrams

The Extended Entity-Relationship (EER) Model

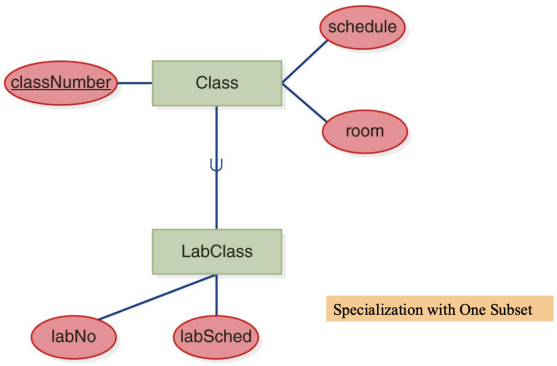
- The ER model is sufficient to represent data needs for traditional applications in business.
- The database used for geographical information systems, search engines, data mining, multimedia, and many other sophisticated applications must be capable of representing more semantic information than the standard ER models can express.
- The extended entity-relationship (EER) model expands the ER model to incorporate various types of abstraction and to express constraints more clearly.
- Additional symbols have been added to standard ER diagrams to create EER diagrams that express these concepts.

Generalization and Specialization

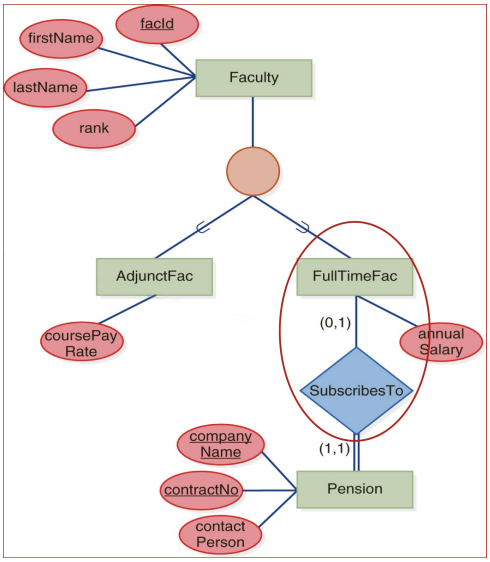
- The process of generalization and its inverse, specialization, are two abstractions that are used to incorporate more meaning in EER diagrams.
- When we have an entity set (Faculty) contains one or more subsets (AdjunctFac, FullTimeFac) that have special attributes or that participate in relationships that other members of the same entity set do not have.



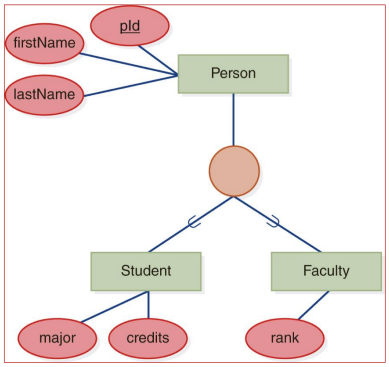
- The specialization circle is sometimes referred to as "is a relationship".



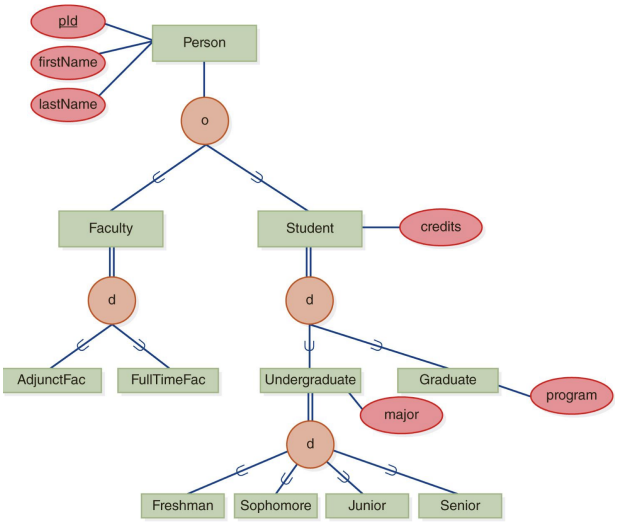
- Subsets may also participate in local relationships that do not apply to the superset or to other subsets in the same hierarchy.
- For example, only full-time faculty members can subscribe to a pension plan.



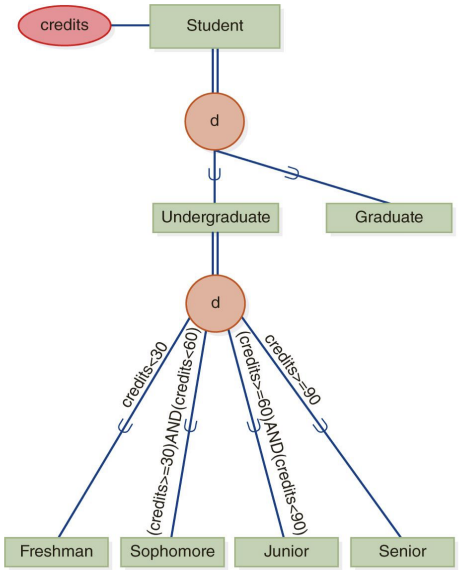
- Set hierarchies can also be created by recognizing that two or more sets have common properties and identifying a common superset for them, a process called generalization.



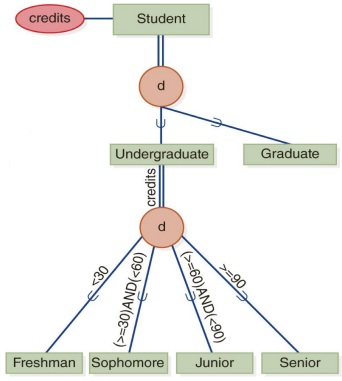
- Subsets can be overlapping, which means the same entity instance can belong to more than one of the subsets.
- Subsets can be disjoint, which means they have no common members. This is referred to as the disjointness constraint.
- This is expressed by placing an appropriate letter, d or o, in the specialization circle.



- A specialization also has a completeness constraint, which shows whether every member of the entity set must participate in it.
- If every member of the superset must belong to some subset, then the specialization is total.
- If some superset members can be permitted to not belong to any subset, the specialization is partial.
- For example, because every student must be either undergraduate or graduate, we have total specialization.

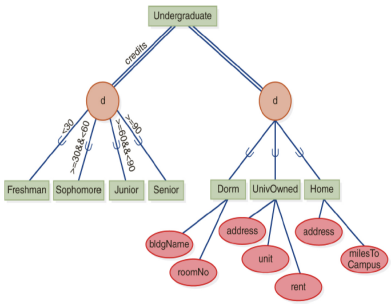


- In some specialization hierarchies, it is possible to identify the subset that the entity belongs to by examining a specific condition or predicate for each subset.
- This is called a predicate-defined specialization.
- In an EER diagram, we can write the defining predicate on the line from the specialization circle.
- Example: In the specialization of Undergraduate, if the predicate credits <30 is true, the student belongs to the freshman subclass, while if the predicate credits >= 30 AND credits <60 is true, the student is a sophomore, and so on.
- The specialization of an entity into subclass is based on the value of a specific attribute.
- A single attribute's value determines the subclass to which an entity belongs.

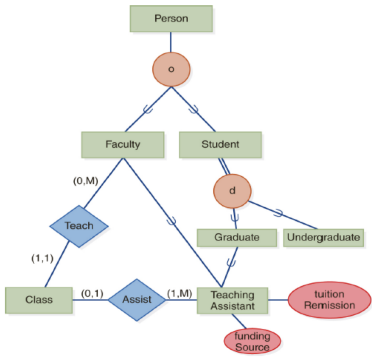


- Specialization that are not predicate-defined are said to be user-defined because the user is responsible for placing the entity instance in the correct set.
- In the previous example, the specialization of Person into Faculty and Student; Faculty into AdjunctFac and FullTime; and Student into Undergraduate and Graduate are all user-defined.
- A database designer could decide that it is important to specialize students by their residence status as well.
- The class year specialization and the residence specialization are independent of each other, and both can appear on the same diagram EER.

- The EER model allows us to give the same superset more than one specialization.

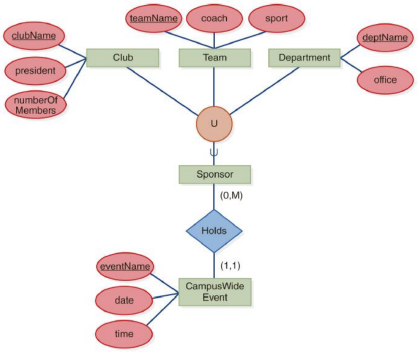


- Sometimes the entity set can be a subset of two or more supersets.
- Such a set is called a shared subset, and it has multiple inheritance from its supersets.
- Members of the TeachingAssistant subset inherit all the attributes of Faculty and of Graduate, and of the supersets of those sets, Student Person, in addition to having their own distinguishing attributes of fundingSource and tuitionRemission.



Union

- Suppose we want to represent sponsors of campus-wide events, which can be sponsored by teams, departments, or clubs.
- If a campus-wide event must have exactly one sponsor, and it can be a team, a department, or a club, we can form a union of Team, Department, and Club, which is then the set of all of these entities.



- Sponsor is a subset of the union representing just one of the many roles that each of these entities might have.
- The attributes of a particular sponsor will depend on which type it is.

- The line connecting sponsor to its union circle is single, indicating a partial category.
- This means that not every Club, Team, or Department has to be a Sponsor.
- CampusWideEvent is a union of Concert or Fair.
- This indicates that every campus-wide event is either a concert or a fair.
- The line connecting CampusWideEvent to the union circle is double, indicating a total category, which means every Concert and Fair must be a member of CampusWideEvent.

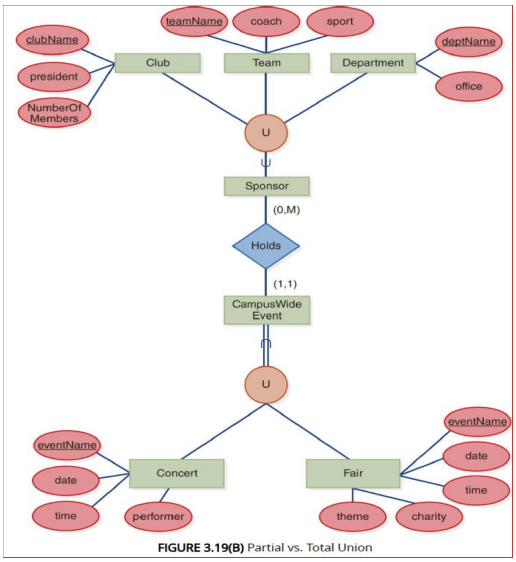
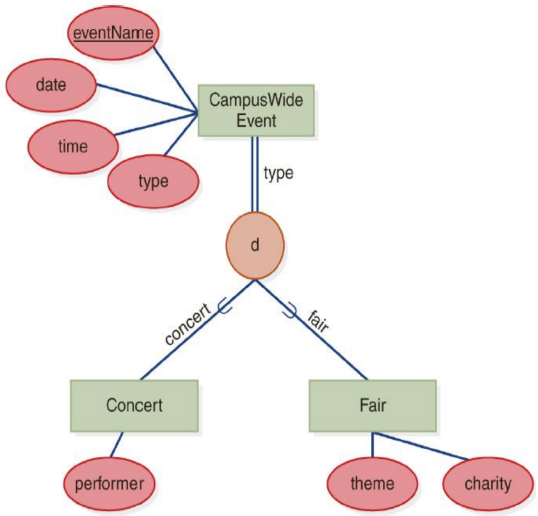
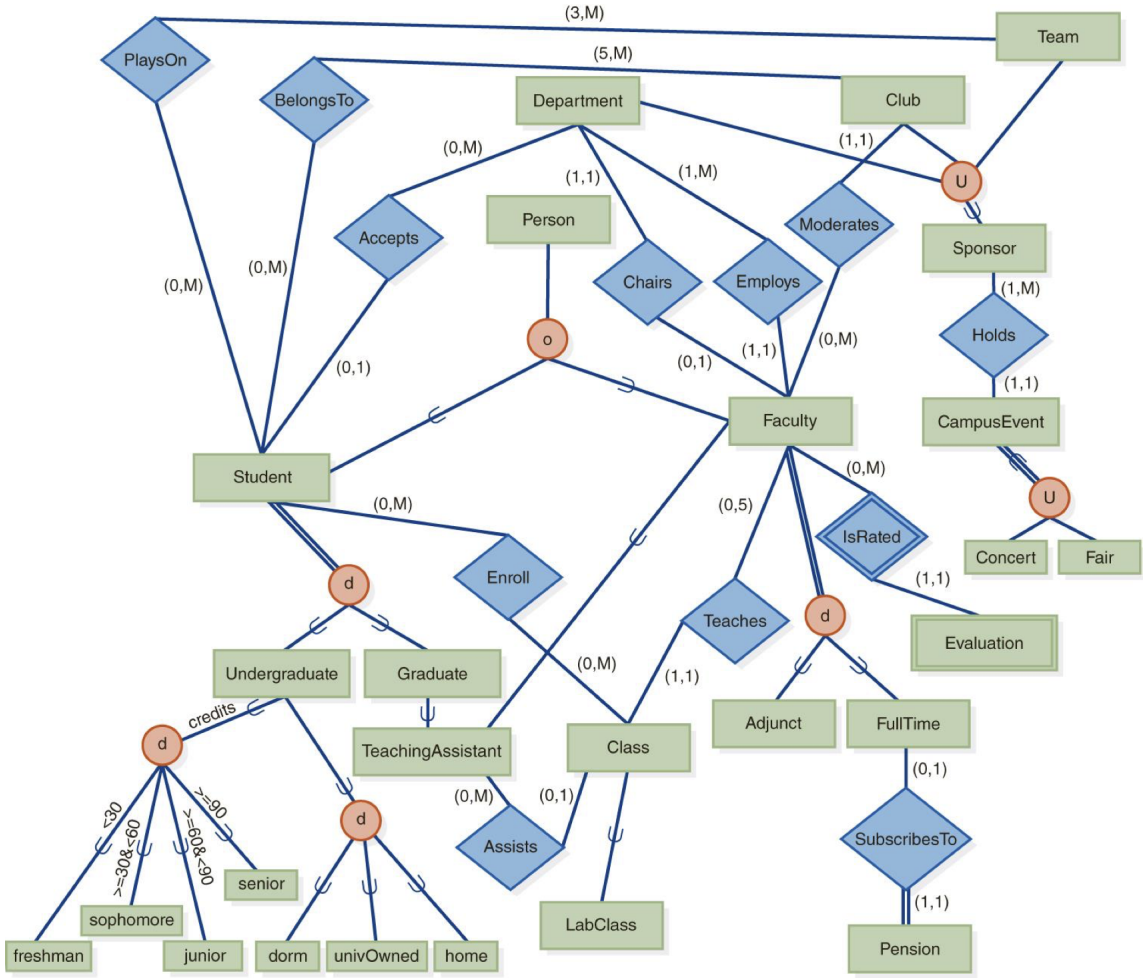


FIGURE 3.19(B) Partial vs. Total Union

- If a union a total and the superclasses share many attributes, it be preferable to use specialization.
- The figure below shows a specialization of campus-wide events into two disjoint subsets, instead of using the union of concerts and fairs into campus-wide events.



Sample EER Diagram



Summary

- Extended Entity-Relationship (EER) Model
  - Enhances the ER model to represent complex relationships found in applications like GIS, search engines, and data mining.
  - Introduces new symbols in ER diagrams for abstraction and constraints.
- Generalization and Specialization
  - Specialization: A single entity set is divided into subsets with specific attributes or relationships.
    - Example: Faculty → AdjunctFaculty, FullTimeFaculty.
    - Uses a specialization circle (sometimes called an "is-a" relationships).
    - Constraints:
      - Disjoint (d): An entity belongs to only one subset.
      - Overlapping (o): An entity can belong to multiple subsets.
      - Total specialization: Every superset entity must belong to a subset.
      - Partial specialization: Some superset entities may not belong to any subset.
    - Predicate-defined specialization: Subsets are determined by attribute conditions.

        · Students → Freshman (credits <30), Sophomore (credits 30-60).
    * User-defined specialization: The user manually assigns entities to subsets.
  - Generalization: Opposite of specialization; multiple entity sets are combined into a higher-level superset.
    * Example: AdjunctFaculty and FullTimeFaculty → Faculty.
  - Multiple Specializations: An entity set can be specialized in more than one way.
    * Example: Students specialized by class year (Freshman, Sophomore) and residence status (On-campus, Off-campus).
  - Shared Subsets (Multiple Inheritance): A subset can inherit attributes from multiple supersets.
    * Example: TeachingAssistant ⊂ Faculty and GraduateStudent (inherits attributes from both).
- Union (Category)
  - Combines multiple entity sets into a single higher-level set.
    * Example: Sponsor ⊂ (Team, Department, Club).
  - Partial category: Not every entity in the supersets has to be part of the union.
    * Example: Not every club, team, or department is a sponsor.
  - Total category: Every instance in the union must belong to the superset.
    * Example: CampusWideEvent ⊂ (Concert, Fair) (every event is either a concert or fair).
  - If supersets share many attributes, specialization is preferred over union.

Lectures 10 - 12

Outline

- Introduction to the Relational Model
- Translating ER Diagrams to Relational Schemas
- Integrity Constraints
- Case Studies for Schema Design
- Mapping an Extended Entity-Relationship (ER) Model to a Relational Schema
- EER to Relational Mapping Example

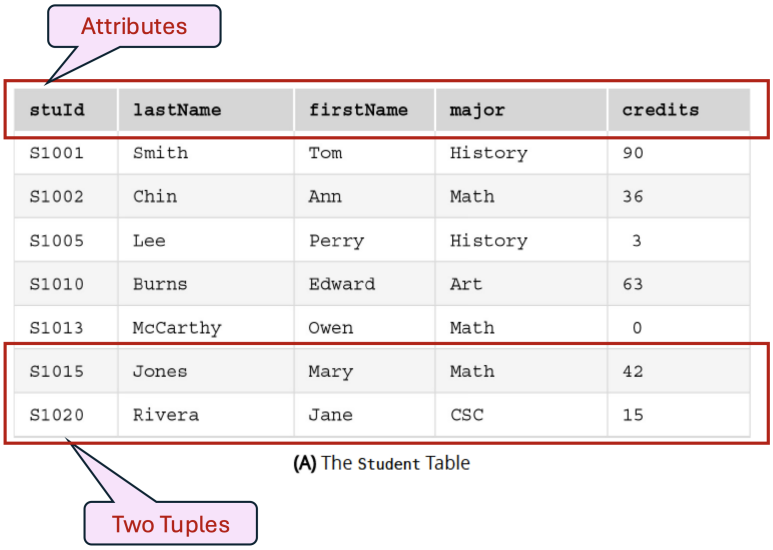
Introduction to the Relational Model

- The relational model was invented by Edgar F. Codd in 1970.
- He introduced the concept in his seminal paper titled "A Relational Model of Data for Large Shared Data Banks", published in the journal Communications of the ACM.
- Codd's relational model revolutionized database management by providing a formal foundation based on mathematical principles (set theory and predicate logic), which became the basis for modern relational database systems.
- Relational Model: A way to represent data using tables (relations) consisting of rows (tuples) and columns (attributes).
- Relation: A table with unique rows.
- Attributes: Columns in a table, each with a domain of values.
- Tuple: A single row in a table.
- Schema: The structure of a table, including its name, attributes, and constraints.

Relational Data Structures

- The data structures used in the relational model are tables with relationships among them.
- The relational model is based on the concept of a relation, which is physically represented as a table or two-dimensional array.
- Using the terms of the entity-relationship model, both entities and relationships are shown using tables.
- A column contains values of a single attribute; for example, the stuId column contains only the IDs of students.

- The domain of an attribute is the set of allowable values for that attribute.
- Each row of the table corresponds to an individual record or entity instance.
- In the relational model, each row is called a tuple.



stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

(A) The Student Table

- A table that represents a relation has the following characteristics:
  - Each cell of the table contains only one value.
  - Each column has a distinct name, which is the name of the attribute it represents.
  - The values in a column all come from the same domain because they are all values of the corresponding attribute.
  - Each tuple or row is distinct; there are no duplicate tuples.
  - The order of tuples or rows in immaterial.
- The structure of the table, together with a specification of the domains and any other restrictions on possible values, shows the intention of the relation, also called the relation schema.
- We can represent relation schemas by giving the name of the relation, followed by the attribute names in parentheses, as:
  - Student(stuID, lastName, firstName, major, credits)

Translating ER Diagrams to Relational Schemas

- Mapping Entities:
  - For each entity in the ER diagram, create a table.
  - Include all simple attributes of the entity as columns.
  - Identify the primary key.
- Example:
  - Entity: Student with attributes StudentID, firstName, lastName, credits, major.
  - Table: Student(StudentID, firstName, lastName, credits, major)

Degree and Cardinality

- The number of columns in a table is called the degree of the relation.
- The degree of a relation never changes unless the structure of the database is changed.
- By contrast, the number of rows in a table, called the cardinality of the relation, changes as new tuples are added or old ones are deleted.

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edwards	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	Comp. Sci.	15

The student table has a degree of 5, and a cardinality of 7.

Relation Keys

- A superkey for a relation is an attribute or a set of attributes that uniquely identifies a tuple.
- A candidate key is a superkey such that no proper subset is a superkey.
- A primary key is the candidate key that is chosen to uniquely identify tuples.
- A foreign key is an attribute or combination of attributes of a relational that is the primary key of some relation.
- Foreign keys are very important in the relational model because they are used to represent logical connections between relations.
- The attribute departmentName, which is a primary key in the Department relation, is a foreign key in Employee.
  - Employee(empId, lastName, firstName, departmentName)
  - Department(departmentName, office)

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	Comp. Sci.	Instructor
F110	Bryne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	Comp. Sci.	Professor

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

  - Super key: {facId, name}, {classNumber, facId}
  - Candidate key: {facId}, {classNumber}
  - Primary key: {facId}, {classNumber}
  - Foreign key: {facId}

Integrity Constraints

- It is important to preserve the integrity of data in a database.
- The term integrity refers to the correctness and internal consistency of the data in the database, by now allowing users to enter data that would make the database incorrect.
- The relational model allows us to define integrity constraints (ICs), which are rules or restrictions that apply to all instances of the database.
- Part of the job of a database management system is to enforce the integrity constraints - to ensure that any data entered creates a legal instance of the database.
- The restrictions on the set of values allowed for attributes of relations are called domain constraints. Domain constraints can be enforced in SQL by defining the domain for an attribute when we create a table.
- An alternative is to use one of the built-in data types and to add a constraint called the CHECK option, which allows us to specify that the value of an attribute must obey a certain condition.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,      – Ensures unique identifier
    Name VARCHAR(50) NOT NULL,      – Name must be a non-null string
    Age INT CHECK (Age BETWEEN 18 AND 30),      – Age must be between 18 and 30
    Email VARCHAR(100) UNIQUE,      – Ensures email is unique
    Gender CHAR(1) CHECK (Gender IN ('M', 'F'))      – Only 'M' or 'F' allowed
    GPA DECIMAL(3,2) CHECK (GPA BETWEEN 0.00 AND 4.00)      – GPA must be in range
);
```

- A primary key constraint, called entity integrity, states that in a relation no attribute of a primary key can have a null value.
- In SQL, we can identify the primary key using a primary key constraint when we create the table. The system will then enforce both the non-null and uniqueness constraints automatically.
- For candidate keys, most systems allow us to specify both a uniqueness constraint and a not null constraint.
- An integrity rule called referential integrity applies to foreign keys.
- Referential integrity states that, if a foreign key exists in a relation, then either the foreign key value must match the primary key value of some tuple in its home relation, or the foreign key value must be completely null.
- SQL allows us to specify foreign key constraints when we create a table.

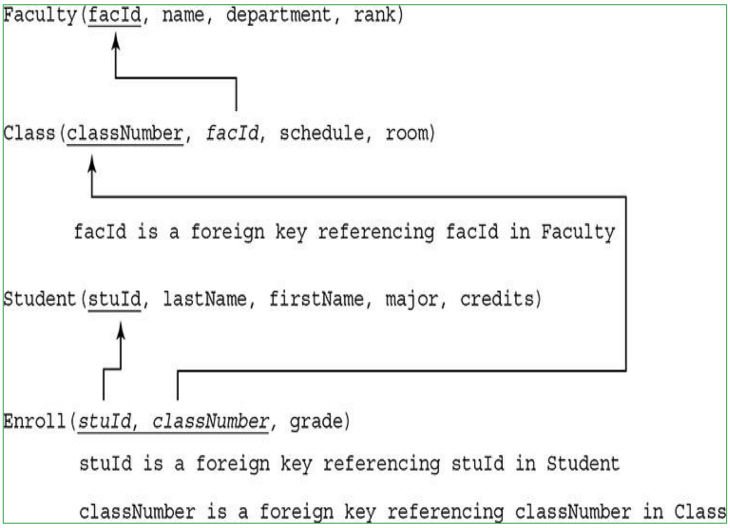
```
CREATE TABLE Departments (
DeptID INT PRIMARY KEY,      – Primary Key
DeptName VARCHAR(50) NOT NULL,      – Department Name
);
```

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,      – Ensures unique identifier
    Name VARCHAR(50) NOT NULL,      – Name must be a non-null string
    Age INT CHECK (Age BETWEEN 18 AND 30),      – Age must be between 18 and 30
    DeptID INT,      – Foreign Key Reference
    FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)      – Foreign Key Constraint
);
```

- There are several other types of constraints, referred to as general constraints or semantic integrity constraints, which we will discuss later.

Representing Relational Database Schemas

- A relational database schema can have any number of relations.
- We can represent relation schemas by giving the name of each relation, followed by the attribute names in parentheses, with the primary key underlined.
- The database schema also includes domains, views, character sets, constraints, stored procedures, authorizations, and other related information.



- The standard way of showing foreign is by drawing arrows from the foreign keys to the primary keys they refer to.
- We will also use italics to indicate that an attribute is a foreign key or part of a foreign key.

Mapping an Entity-Relationship Model to a Relational Schema

- Both entities and relationships must be represented in the relational schema.
  - The strong entity sets shown as rectangles become relations represented by base tables.
    - The table name is the same as the entity name.
  - For strong entity sets, non-composite, single-valued attributes, represented by simple ovals, become attributes of the relation, or column headings of the table.
    - Department(deptName, deptCode, office)
    - Student(stuId, lastName, firstName, credits)
    - Faculty(facId, lastName, firstName, rank)
  - For ER attributes the are composites, we can make a column for each of the simple attributes that form the composite, or we can choose the leave the composite as a single attribute.
    - For example, if we had a composite attribute called Address that consisted of street, city, state, zipCode, we could choose to make columns for each of these attributes, or we could keep the entire address as a single column, Address.
    - The first choice would make it easier to select rows with a certain city or state.
    - In the university example, classNumber is a composite consisting of deptCode, courseNumber, and section.
    - We could use the first method, showing each of the sub-attributes, as:
      - Class1(deptCode, courseNumber, section, schedule, room)

    - Or we can choose to leave the composite as a single attribute. Therefore, we form the Class table as:
        - Class(classNumber, schedule, room)
- Multi-valued attributes pose a special problem when converting to a pure relational model, which does not allow multiple values in a cell.
    - The solution is to remove them from the table and create a separate relation (table) in which we put the primary key of the entity along with the multi-valued attribute.
    - Ex: If the attribute author can have multiple values
        - Textbook(isbn, title, publisher)
        - TextAuthors(isbn, author)
    - The key of this new table becomes the foreign keys of the original table and the multi-valued attribute. The key of the original table becomes a foreign key in the new table.
    - If there are several multi-valued attributes in the original table, we must create a new table for each one.
    - It is convenient to name the new table(s) using the plural form of the multi-valued attribute name.
- Weak entity sets are also represented by tables, but they require additional attributes.
    - A weak entity is dependent on another (owner) entity and has no primary key consisting of just its own attributes.
    - We use the combination of the owner's primary key and the discriminant as the primary key of the weak entity.
    - For example, the weak entity Evaluation, is given the primary key of its owner entity, Faculty, resulting in the table.
        - Evaluation(facId, date, rater, rating)
- If A and B are strong entity sets and have a binary relationship that is one-to-many, we place the key of A (the one side) in the table for B (the many side), where it becomes a foreign key.
    - Teach is a one-to-many relationship sets connecting the strong entity sets Faculty and Class.
    - We use a foreign key to represent this relationship, by placing facId, the key of the "one" side, Faculty, in the table for the "many" side, Class.
        - Class(classNumber, schedule, room, *facId*)
    - If we had tried to represent the relationship Teach in the opposite way, by placing the classNumber in the Faculty table, it would not work.
    - Because a faculty member teach many courses, a row in the Faculty table would need several values in the classNumber cell, which is not allowed.
- All strong entity sets that have a one-to-one relationship should be examined carefully to determine whether they are the same entity. If they are, their attributes should be combined into a single entity.
    - If A and B are truly separate entities having a one-to-one relationship, then we can put the key of either relation in the other table to show the connection (i.e., either put the key of A in the B table, or vice versa, but not both).
    - Ex: Let us assume students can have parking permit to part one car on campus. We could add a Car entity, which would have a one-to-one relationship with Student. Each car belongs to exactly one student and each student can have at most one car. The initial schema for Car might be:
        - Car(permitNumber, licNo, make, model, year, color)
        - To store the one-to-one relationship with Student, we could put stuID in the Car table.
        - Car(permitNumber, licNo, make, model, year, color *stuId*)
        - Alternatively, we could add permitNumber to the Student table - but not both.

- For a binary relationship, the only time that we cannot use a foreign key to represent the relationship set is the many-to-many case.
  - Here, the connection must be shown by a separate relationship table.
  - Ex: The Enroll relationship is a many-to-many relationship. The table that represents it must contain the primary keys of the associated Student and Class entities, stuId and classNo, respectively. Because the relationship also has a descriptive attribute, grade, we include that as well.
  - The relationship table is:
    - Enroll(stuId, classNumber, grade)
- When we have ternary or n-ary relationship involving three or more entity sets, we can construct a table for the relationship, in which we place the primary keys of the related entities.
  - If the ternary or n-ary relationship has a descriptive attribute, it goes in the relationship table.
  - Ex: The ternary relationship, Course-Faculty-Textbook we could represent it by the table:
    - Course-Faculty-Textbook(facId, courseNo, isbn)
  - We would choose the combination of courseNo and facId as the primary key.
- A recursive relationship representation depends on the cardinality.
  - If the cardinality is many-to-many, we must create a relationship table.
  - If it is one-to-one or one-to-many, the foreign key mechanism can be used.
  - For example, if we wanted to represent a one-to-one Roommate relationship and we assume each student can have at most one roommate, we could represent the one-to-one recursive relationship by adding an attribute, roommateStuId, to the Student table, where it would be a foreign key, as in:
    - Student3(stuId, lastName, firstName, major, credits, *roommateStuId*)

- For entity sets and relationships that are not part of generalizations or unions, the mapping proceeds as for the ER model.
  - Map strong entity sets to tables having a column for each single-valued, non-composite attribute.
  - For composite attributes, create a column for each component and do not represent the composite. Alternatively, create a single column for the composite and ignore the components.
  - For each multi-valued attribute, create a separate table having the primary key of the entity, along with a column for the multi-valued attribute. Alternatively, if the maximum number of values for the attribute is known, create multiple columns for these values.
  - For each weak entity set, create a table that includes the primary key of the owner entity along with the attributes of the weak entity set.
  - For one-to-many binary relationships, place the primary key of the one side in the table of the many side, creating foreign key.
  - For one-to-one binary relationships, place either one of the primary keys in the table for the other entity.
  - For many-to-many binary relationships, and for all higher-level relationships, create a relationship table that consists of the primary keys of the related entities, along with any descriptive attributes of the relationship.

- HasMajor relationship
  - To represent the 1:M HasMajor relationship connecting Department to Student, we place the primary key of the "one" side, Department, int he table for the "many" side, Student, changing its name to major.
  - Recall the value of major is the same as deptName in Department.

  - We cannot do the opposite, placing the stuId in the Department table, because a department has many student majors and we cannot have an attribute with multiple values.
  - Student(<u>stuId</u>, lastName, firstName, credits, major)

- Teach relationship
  - Teach is a one-to-many relationship that connects Faculty to Class. As discussed, we represent this relationship by placing facId as a foreign key in Class.

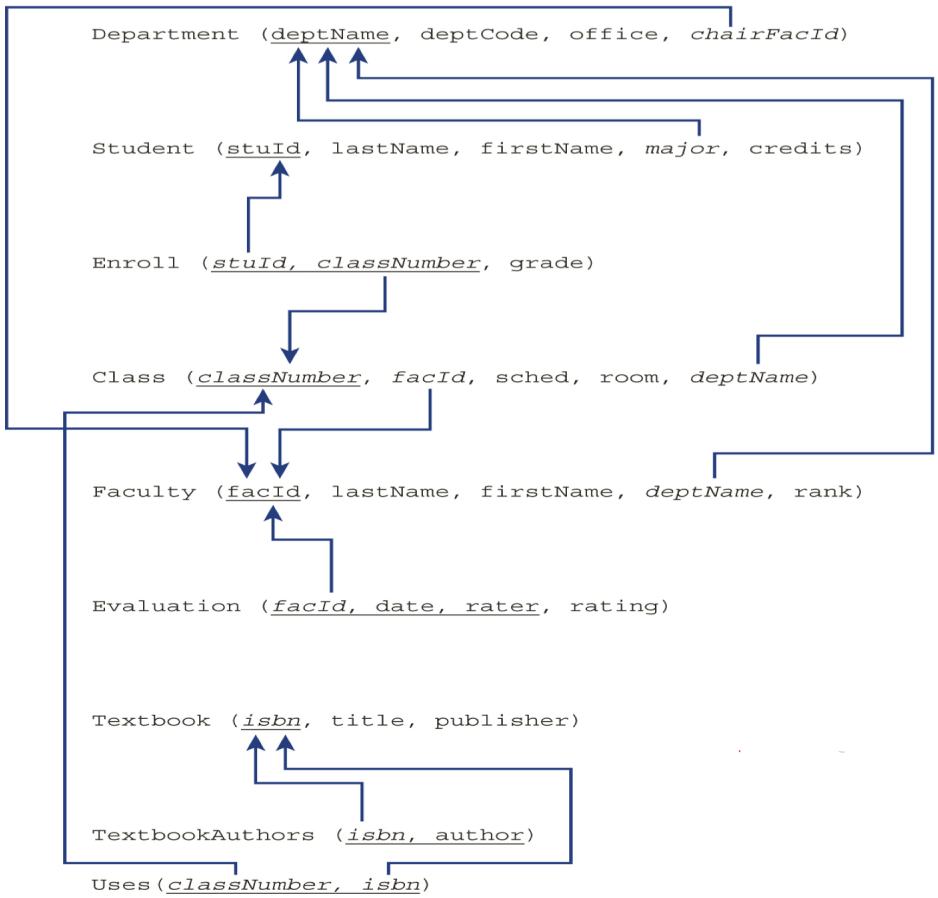
- Offers relationship
  - Offers is a one-to-many relationship that connects Department to Class. We represent this relationship by putting the key of Department, deptName, in the Class table, changing it to:
    * Class(<u>classNumber</u>, schedule, room, *facId*, *deptName*)

- Employ relationship
  - Employ is a one-to-many relationship that represents the association between Department and Faculty. We use the foreign key mechanism, adding deptName as an attribute in the Faculty table, making that table.
    * Faculty(<u>facId</u>, lastName, firstName, rank, *deptName*)

- Chairs relationship
  - Chairs is a one-to-one relationship that connects Department to Faculty. We will push the primary key of Faculty into Department as a foreign key, changing its name slightly for clarity, making the Department schema.

- isRated relationship
  - The isRated relationship connects the weak entity Evaluation to its owner, Faculty.
  - It is already represented by having the primary key of the owner entity in the weak entity's table as part of its primary key.

- Relational Schema for University:



- The techniques discussed previously for mapping an ER model to a relational model apply to the EER model, but they must be extended to take into account the additional features of the EER mode, namely specialization / generalization and union.
- Mapping EER Set Hierarchies to Relational Tables
  - Method 1: Create a table for the superset and one for each of the subsets, placing the primary key of the superset in each subset table.
    - We create three tables, Faculty, AdjunctFac, and FullTimeFac.
    - Both AdjunctFac and FullTimeFac tables have columns for their own attributes, plus columns for the primary key of the superset, facId, which functions both as the primary key and a foreign key in those tables.
    - Faculty(facId, lastName, firstName, rank)
    - AdjunctFac(facId, coursePayRate)
    - FullTimeFac(facId, annualSalary)
  - Method 2: Create tables for each of the subsets and no table for the subset. Place all of the attributes of the superset in each of the subset tables.
    - For this example, we would have tables for AdjunctFac and FullTimeFac, and none for Faculty.
    - AdjunctFac(facId, lastName, firstName, rank, coursePayRate)
    - FullTimeFac(facId, lastName, firstName, rank, annualSalary)
  - Method 3: Create a single table that contains all the attributes of the superset, along with the attributes of all subsets.
    - For our example, we could create the table:

    * AllFac(facID, lastName, rank, annualSalary, coursePayRate)
  - A variation of Method 3 is to add a "type field" to the record, indicating to which subset the entity belongs. For attribute-defined disjoint specializations, this is simply the defining attribute. For overlapping specializations, we could add a type field for each specializations, indicating whether the entity belongs to each of the subsets.
  - There are trade-offs to be considered for each of these methods.
    * The first works for all types of subsets, regardless of whether the specialization is disjoint or overlapping; partial or total; and attribute-defined or user-defined.
    * The second method handles queries about the subsets well because one table has all the information about each subset. However, it should not be used if the subsets are overlapping because data about the instances that belong to more than one subset will be duplicated. It cannot be used if the specialization is partial because the entity instances that do not belong to any subset cannot be represented at all using this method.
    * The third method allows us to answer queries about al the subsets and the superset without doing joins, but it will result in storing many null values in the database.

- Mapping Unions
  - To represent unions (categories), we create a table for the union and individual tables for each of the supersets, using foreign keys to connect them.
  - A problem that often arises with mapping unions is that the supersets can have different primary keys.
  - The solution is to create a surrogate key that will be the primary key of the union. it will be a foreign key in the tables for each of the supersets.
  - For the Sponsor example, we create a surrogate key for Sponsor, which we call sponsorId, and we insert that attribute as a foreign key in the tables for Club, Team, and Department.
  - We also add SponsorType to indicate whether the sponsor is a club, team or department.
    * Sponsor(sponsorId, sponsorType)
    * Club(clubName, president, numberOfMembers, *sponsorId*)
    * Team(teamName, coach, sport, *sponsorId*)
    * Department(deptName, office, *sponsorId*)
  - If the primary keys of the supersets are the same, it is not necessary to create a surrogate key.
  - For the CampusWideEvent example, the schema will be:
    * CampusWideEvent(eventName, eventType)
    * Concert(eventName, date, time, performer)
    * Fair(eventName, date, time, theme, charity)

EER to Relational Mapping Example

- We have a hierarchy for Person, with subsets Student and Faculty, each of which itself has subsets.
- Our first decision is not to represent the Person superset because it does not participate as a whole in any relationships.
- We create separate tables for Student and Faculty, repeating the common Person attributes in each, which is Method 2.
  - Faculty(facId, lastName, firstName, rank)
  - Student(stuId, lastName, firstName, credits)
- Create tables for both undergraduate and graduate students, which is Method 1.

- We change the name of the stuId field slightly to reduce confusion.
  - Undergraduate(undergradStuId, major)
  - Graduate(gradStuId, program)
  - Faculty(facId, lastName, firstName, rank)
  - AdjunctFac(adjFacId, coursePayRate)
  - FullTimeFac(fullFacId, annualSalary)
- The Undergraduate subset has subsets for class years and for residences, but the one for class years is attribute-defined.
- It does not seem necessary to break up the Undergraduate table by class year. If we need to record any attributes specific to a class year, such as a thesis advisor for seniors, we add those to the Undergraduate table.
- Similarly, for residence, the residence type and address might be sufficient, so we use Method 3 and we change the undergraduate table to be:
  - Undergraduate(undergradStuId, major, thesisAdvisor, resType, address)
- Graduate has a single subset, TeachingAssistant, and we decide to represent that specialization using Method 3, by adding its two attributes to the Graduate table, changing the table to:
  - Graduate(gradStuId, program, tuitionRemission, fundingSource)
- The diagram shows that TeachingAssistant is also a subclass of Faculty, so we put facId in as an attribute for those graduate students who are TAs.
  - Graduate(gradStuId, program, tuitionRemission, fundingSource, facId)

Summary

- Representing Relational Database Schemas
  - A relational database schema consists of multiple relations, defined by their names and attributes.
  - Primary keys are underlined, while foreign keys are italicized.
  - Other schema elements include domains, views, constraints, stored procedures, and authorizations.
  - Foreign keys are visually represented using arrows pointing to the corresponding primary keys.
- Mapping ER Models to Relational Schemas
  - Entities → Tables: Strong entity sets become base tables, with attributes forming columns.
  - Composite Attributes: Can be broken into separate columns or kept as a single column.
  - Multi-Valued Attributes: Require a separate table containing the entity's primary key and the multi-valued attribute.
  - Weak Entities: Represented using a table with the owner's primary key included as part of its primary key.
  - One-to-Many Relationships: The primary key of the "one" side is placed in the table for the "many" side.
  - One-to-One Relationships: A foreign key is added to either of the entity tables.
  - Many-to-Many Relationships: A separate relationship table is created using the primary keys of both entities.
  - Ternary and N-ary Relationships: Require a separate table containing the primary keys of related entities.

  - Recursive Relationships: One-to-many and one-to-one relationships can use foreign keys, while many-to-many requires a separate table.

- Mapping EER to Relational Schemas
  - Specialization / Generalization
    * Create a table for the superset and separate tables for each subset (foreign key points to superset).
    * Create only subset tables, repeating superset attributes.
    * Create a single table including attributes from all subsets, often resulting in NULL values for unused attributes.
  - Union (Category) Representation
    * Create a table for the union and separate tables for each superset, using foreign keys to link them.
    * If supersets have different primary keys, introduce a surrogate key.

Lectures 13 - 14

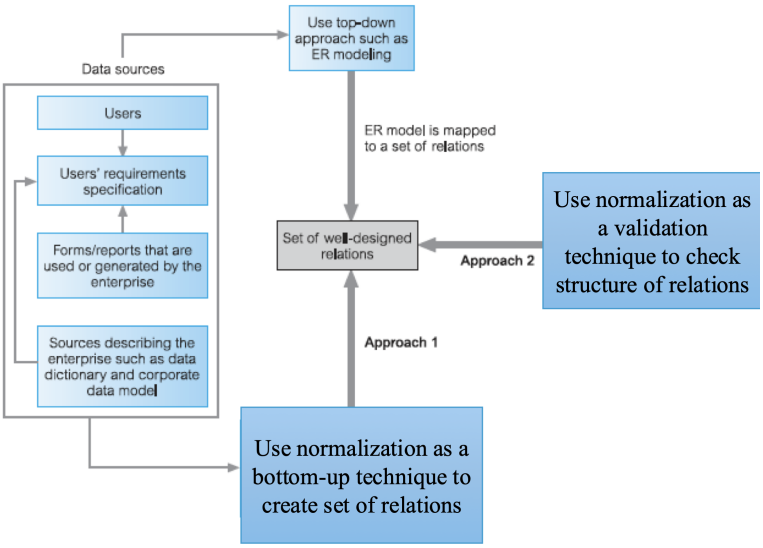
Outline

- The Purpose of Normalization
- How Normalization Supports Database Design
- Data Redundancy and Update Anomalies
- Functional Dependencies
- The Process of Normalization
- Normal Forms

The Purpose of Normalization

- What is Normalization?
  - Normalization is a process in database design that organizes data to reduce redundancy and improve data integrity.
  - Normalization is based on examining the relationships (called functional dependencies) between attributes.
  - Normalization uses a series of tests (described as normal forms) to help identify the optimal grouping for these attributes.
- Normalization serves several important purposes in database design:
  - Eliminates Data Redundancy: Reduces unnecessary duplication of data, saving storage space and improving efficiency.
  - Ensures Data Integrity and Consistency: Prevents anomalies that can occur when inserting, updating, or deleting records.
  - Improves Query Performance: Organized data allows for faster and more efficient data retrieval.
  - Simplifies Data Maintenance: Reduces errors and inconsistencies when modifying data.
  - Supports Scalability: A well-normalized database can handle growth and complex queries more effectively.

How Normalization Supports Database Design



- How They Work Together (Approach #2)
  - Start with an ER Diagram (Conceptual Model)
  - Map the ER Model to a Relational Schema (Logical Model)
  - Apply Normalization (Schema Refinement)

Aspect	ER Diagram	Mapping to a Relational Model	Normalization
Purpose	Conceptually represents entities, attributes, and relationships.	Convert the ER model into a relational schema with tables, primary keys, and foreign keys.	Refines the relational schema to reduce redundancy and ensure data integrity.
Focus	Focuses on high-level relationships between data entities.	Focuses on converting conceptual entities into tables with relationships and constraints.	Focuses on structural optimization of tables to remove anomalies.
Methodology	Identifies entities, attributes, and relationships. Defines cardinality (1:1, 1:N, M:N).	Entities → Tables, Attributes → Columns, Relationships → Foreign keys, Weak entities → Tables with composite keys, M:N relationships → Separate tables.	Applies 1NF, 2NF, 3NF, BCNF to remove redundancy and anomalies. Uses functional dependencies to ensure consistency.
Outcome	Produces and ER diagram showing data structure and relationships.	Produces an initial relational schema with primary keys, foreign keys, and constraints.	Produces a normalized schema with improved consistency and efficiency.

Data Redundancy and Update Anomalies

- The problems associated with unwanted data redundancy.
  - Staff(staffNo, sName, position, salary, branchNo)
  - Branch(branchNo, bAddress)
  - StaffBranch(staffNo, sName, position, salary, branchNo, bAddress)
- **Staff:**

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

- **Branch:**

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

- **StaffBranch:**

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

- Which design is better, using two tables (Staff, Branch) or one table (StaffBranch)?
  - In the StaffBranch relation there is redundant data; the details of a branch are repeated for every member of staff located at that branch.
  - In the branch details appear only once for each branch in the Branch relation, and only the branch number (branchNo) is repeated in the Staff relation to represent where each member of staff is located.
- Relations that have redundant data may have problems called update anomalies, which are classified as:
  - insertion,
  - deletion, or
  - modification.

Insertion Anomalies

- To insert the details of new members of staff into the StaffBranch relation, we must include the details of the branch (branchNo, bAddress) at which the staff are to be located. What if we entered a wrong address?
- We cannot enter a tuple of a new branch into the StaffBranch relation with a null for the staffNo.

Deletion Anomalies

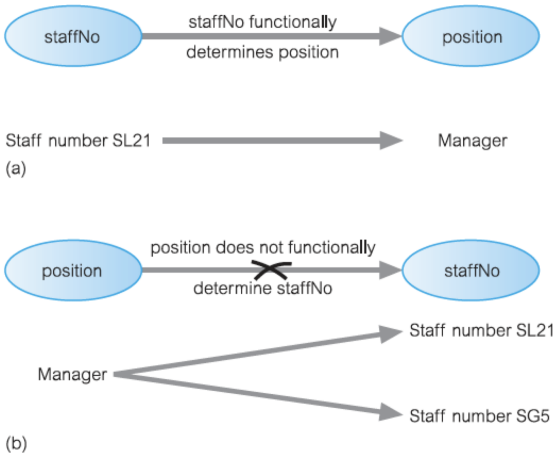
- If we delete the tuple for staff number SA9 (Mary Howe) from the StaffBranch relation, what will happen to branch B007 data?

Modification Anomalies

- If we want to change the value of one of the attributes of a particular branch in the StaffBranch relation – for example, the address for branch number B003 – we must update the tuples of all staff located at that branch.

Functional Dependencies

- An important concept associated with normalization is functional dependency.
- A functional dependency (FD) is a constraint between two sets of attributes in a relational database. It expresses a relationship where the value of one attribute (or a group of attributes) uniquely determines the value of another attribute.

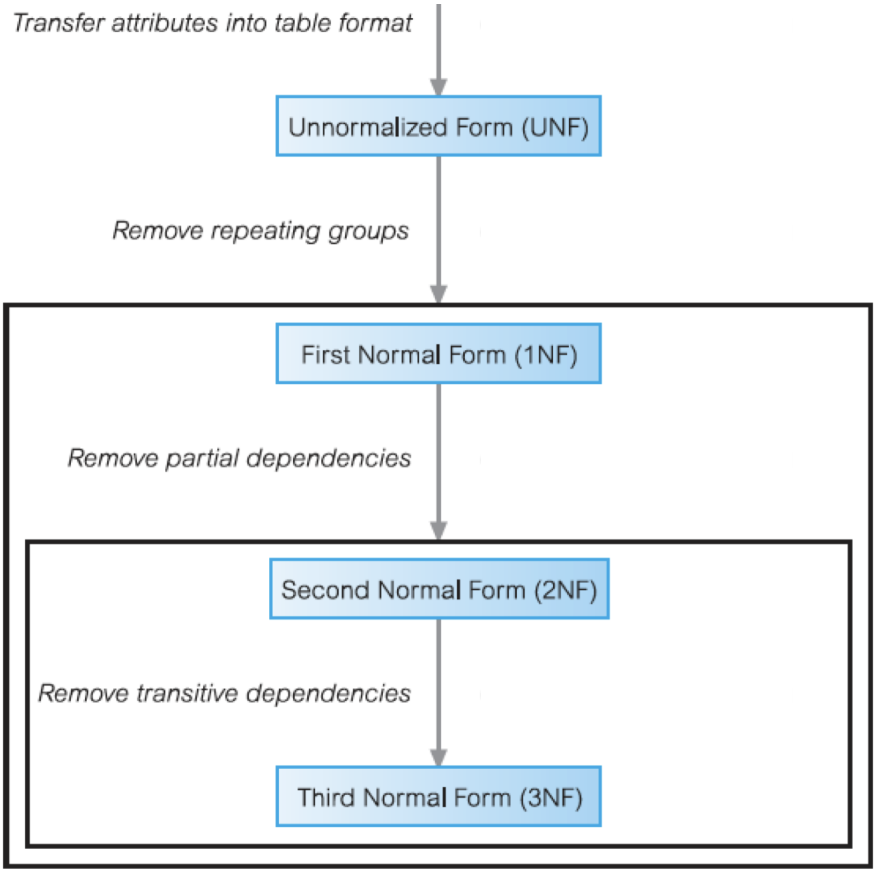


- If A and B are attributes of relation R, B is functionally dependent on A (denoted A Ⓡ B), if each value of A is associated with exactly one value of B. (A and B may each consist of one or more attributes.)
- A functional dependency A Ⓡ B is a full functional dependency if removal of any attribute from A results in the dependency no longer existing.
- A functional dependency A Ⓡ B is a partial dependency if there is some attribute that can be removed A and yet the dependency still holds.
  - Example: staffNo, sName Ⓡ branchNo
  - This is a partial dependency, because branchNo is also functionally dependent on a subset of (staffNo, sName), namely staffNo.
- Transitive Dependency: A condition where A, B, and C are attributes of a relation such that if A Ⓡ B and B Ⓡ C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).
  - Example: staffNo Ⓡ sName, position, salary, branchNo, bAddress
  - branchNo Ⓡ bAddress
  - The staffNo attribute functionally determines the bAddress via the branchNo attribute and neither branchNo nor bAddress functionally determines staffNo.
- Identifying Functional Dependencies
  - Use the meaning of each attribute and the relationships between the attributes, or, use sample data is available that is a true representation of all possible data value that the database may hold.
  - Example: We begin by examining the semantics of the attributes in the StaffBranch relation.
  - For the purposes of discussion, we assume that the position held and the branch determine a member of staff's salary.
  - We identify the functional dependencies based on our understanding of the attributes in the relation as:

- staffNo Ⓡ sName, position, salary, branchNo, bAddress
- branchNo Ⓡ bAddress
- bAddress Ⓡ branchNo
- branchNo, position Ⓡ salary
- bAddress, position Ⓡ salary

The Process of Normalization (Continued)

- Normalization is a formal technique used to analyze database relations based on primary keys and functional dependencies.
- It follows a series of rules to structure a database by decomposing relations that violate normalization requirements.
- Initially, three normal forms were introduced: First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).
- Later, Boyce-Codd Normal Form (BCNF) (Codd, 1974) was introduced as a stronger version of 3NF.
- Higher normal forms like Fourth Normal Form (4NF) and Fifth Normal Form (5NF) (Fagin, 1977, 1979) address rare cases.
- Normalization is performed in steps, with each normal form imposing stricter rules to reduce update anomalies.
- While only 1NF is essential for relational databases, progressing to at least 3NF is generally recommended for better data integrity.



First Normal Form (1NF)

- Definition: A relation (table) is in First Normal Form (1NF) if:
  - All attributes (columns) contain atomic (indivisible) values.
  - Each column contains values of a single type (no mixed data types).
  - Each row has a unique identifier (Primary Key).
- How to Achieve 1NF
- To convert a table to First Normal Form (1NF):
  - Eliminate repeating groups: Ensure each column contains only one value per row (no lists or sets).
  - Create separate rows for multiple values: If a column contains multiple values, split them into separate rows.
  - Ensure all columns have atomic values: Avoid storing composite or multi-valued attributes in a single field.
  - Identify a primary key: Ensure each row is uniquely identifiable.
- Example (Before 1NF):

StudentID	Name	Course
101	John	Math, Science
102	Alice	English
103	Bob	Math, Physics

- After Converting to 1NF:

StudentID	Name	Course
101	John	Math
101	John	Science
102	Alice	English
103	Bob	Math
103	Bob	Physics

Second Normal Form (2NF)

- What is 2NF?
- Second Normal Form (2NF) ensures that all non-key attributes are fully functionally dependent on the entire primary key.
- This means that:
  - The table must already be in First Normal Form (1NF).
  - No partial dependencies should exist - each non-key attributes must depend on the whole primary key, not just a part of it.
- When to apply 2NF?
  - When the table has a composite primary key (a key with multiple attributes).
  - If some non-key attributes depend on only part of the primary key instead of the whole key.
  - If the table is prone to update anomalies, meaning redundant data updates may cause inconsistencies.
- How to apply 2NF?

- Identify partial dependencies: Check if any non-key attributes depend only on part of the primary key.
- Remove partial dependencies: Move these attributes to a separate table.
- Create a foreign key relationship: Establish a link between the new table and the original table.

- Example (Before 2NF - In 1NF but not 2NF)

OrderID	ProductID	Product Name	Quantity
1	101	Laptop	2
1	102	Mouse	3
2	101	Laptop	1

- After Applying 2NF (Removing Partial Dependency)

OrderID	ProductID	Quantity
1	101	2
1	102	3
2	101	1

ProductID	ProductName
101	Laptop
102	Mouse

- Step 1: Identify the Primary Key
  - Since each row represents a product in an order, the OrderID alone is not unique.
  - A combination of OrderID + ProductID uniquely identifies each row.
  - So, the composite primary key is: (OrderID, ProductID)
- Step 3: Identify the Functional Dependencies
  - (OrderID, ProductID) → Quantity
    - The quantity of a product depends on both OrderiD and ProductID (i.e., how many of a certain product were ordered in that specific order).
  - ProductID → ProductName
    - The product name is fully dependent on ProductID only.
    - This is a partial dependency because ProductName does not depend on the full composite key (OrderID, ProductID), only on ProductID.
- Step 3: Understanding the Issue
  - The second functional dependency (ProductID → ProductName) violates 2NF because:
  - This causes redundancy - the same product name (e.g., "Laptop") appears multiple times for different orders.
- Step 4: Resolving the Issue (Moving to 2NF)
  - To remove the partial dependency, we split the table into two.

Third Normal Form (3NF)

- What is 3NF?
  - A relation is in Third Normal Form (3NF) if:
    - It is already in 2NF (i.e., no partial dependencies).
    - No non-primary key attribute is transitively dependent on the primary key.

  - Transitive dependency means that a non-primary attribute depends on another non-primary attribute rather than directly on the primary key.

- When to apply 3NF?
  - Even after 2NF, redundancy and update anomalies can still exist due to transitive dependencies.
  - If updating a non-primary attribute requires changing multiple rows, it indicates a transitive dependency, meaning 3NF is required.

- How to apply 3NF?
  - Identify transitive dependencies: If an attribute depends on another non-primary key attribute rather than the primary key.
  - Decompose the relation: Move the transitively dependent attribute to a new table.
  - Link the new table with a foreign key.

- Before 3NF (2NF Table - Property Owner)
  - OwnerName depends on OwnerID, not directly on PropertyID (Primary Key).

PropertyID	OwnerID	OwnerName
P101	CO93	Tony Shaw
P102	CO93	Tony Shaw

- After 3NF (Decomposed into Two Tables)
  - Property table

PropertyID	OwnerID
P101	CO93
P102	CO93

  - Owner table

OwnerID	OwnerName
CO93	Tony Shaw

BCNF

- What is BCNF?
  - BCNF (Boyce-Codd Normal Form) is an advanced version of 3NF, which removes all remaining anomalies related to functional dependencies.
  - A table is in BCNF if:
    * It is already in 3NF.
    * Every determinant is a candidate key (This means every attribute that functionally determines another attribute must itself be a candidate key).
    * Note: A candidate key is a minimal set of attributes that can uniquely identify a row in a table.

- When to apply BCNF?
  - If a functional dependency exists where the determinant is not a candidate key, the table needs BCNF decomposition.

- How to apply BCNF?
  - Identify functional dependencies where a non-candidate key determines other attributes.

- Decompose the relation by creating separate tables.
- Ensure all tables follow the BCNF rule (every determinant must be a candidate key).

- Before BCNF (3NF Table - Course Instructor)
  - Functional Dependency: Instructor → InstructorOffice
  - Issue: Instructor is not a candidate key, meaning a non-key attribute (Instructor) determines another attribute (InstructorOffice).

CourseID	Instructor	InstructorOffice
C101	Prof. A	Room 201
C102	Prof. B	Room 202
C103	Prof. A	Room 201

- After BCNF (Decomposed Tables)
  - Course table

CourseID	Instructor
C101	Prof. A
C102	Prof. B
C103	Prof. A

  - Instructor table

Instructor	InstructorOffice
Prof. A	Room 201
Prof. B	Room 202

Denormalization

- What is Denormalization?
  - Denormalization is the process of intentionally introducing redundancy into a database to improve read performance. It is the opposite of normalization.
  - Instead of strictly following higher normal forms (like 3NF, BCNF, etc.) denormalization merges tables or duplicates data to reduce the need for complex joins, making queries faster.
- When to apply Denormalization?
- Denormalization should be applied only when performance issues arise and after careful analysis. Situations where denormalization helps:
  - Slow Query Performance: If queries involve too many joins, it may slow down data retrieval.
    - Example: A reporting dashboard that fetches aggregated data across multiple tables.
  - Frequent Reads, Rare Writes: If the database has many read operations but few updates, denormalization can help.
    - Example: A customer order history page that loads frequently but updates rarely.
  - Complex Queries Involving Joins: If a system frequently joins multiple tables, denormalization can reduce the number of joins.
    - Example: Instead of joining Orders and Customers, store customer details directly in Orders.
  - Data Warehousing & Analytics: Analytical databases prioritize read performance over normalization.
    - Example: A sales report that pre-aggregates monthly revenue instead of computing it on the fly.

How to Apply Denormalization?

1. Adding Redundant Data
   - Store frequently used data in multiple tables to avoid joins.
   - Example: Instead of joining Customers and Orders, store CustomerName in Orders.
2. Precomputing Aggregations
   - Store calculated values like total sales per customer instead of computing them dynamically.
   - Example: Instead of calculating SUM(OrderAmount), store TotalSpent in the Customers table.
3. Merging Tables
   - Combine tables that are frequently queried together.
   - Example: Instead of separate Orders and OrderDetails, create a single OrderSummary table.
4. Using Indexed Views
   - Create a materialized view (precomputed query results) to speed up data retrieval.
   - Example: A view that stores total orders per product instead of recalculating it on demand.
5. Trade-offs of Denormalization

Pros	Cons
Faster read queries	Increased storage usage
Fewer complex joins	Risk of data inconsistency
Better performance for analytics	Harder to maintain updates

Summary

- Purpose of Normalization
  - Normalization is a process in database design that organizes data to:
    - Eliminate redundancy and save storage space.
    - Ensure data integrity and consistency by preventing anomalies.
    - Improve query performance for efficient data retrieval.
    - Simplify data maintenance and reduce errors.
    - Support scalability for handling growth and complex queries.
- How Normalization Supports Database Design
  - ER Diagram (Conceptual Model) → Defines entities, attributes, and relationships.
  - Relational Schema (Logical Model) → Converts entities into tables with primary and foreign keys.
  - Normalization (Schema Refinement) → Reduces redundancy and ensures integrity through normal forms.
- Data Redundancy and Update Anomalies
  - Redundancy Issue: Storing staff and branch details in a single table leads to duplication.
  - Better Design: Splitting data into separate Staff and Branch tables reduces redundancy.
  - Update Anomalies:
    - Insertion: New branches cannot be added without staff.
    - Deletion: Removing a staff record may delete branch data.
    - Modification: Changing branch details requires updating multiple rows.

- Functional Dependencies
  - Definition: A constraint where one attribute uniquely determines another (e.g., staffNo $\rightarrow$ sName).
  - Types:
    - Full Dependency: Removing an attribute breaks the dependency.
    - Partial Dependency: A subset of attributes still maintains the dependency.
    - Transitive Dependency: A determines B, and B determines C, leading to an indirect dependency.
- Normalization Stages:
  - First Normal Form (1NF): Ensures attributes contain atomic values,e ach column has a single data type, and rows have a unique identifier (Primary Key). Repeating groups are removed.
  - Second Normal Form (2NF): Requires 1NF compliance and removes partial dependencies, ensuring non-key attributes depend on the entire primary key.
  - Third Normal Form (3NF): Eliminates transitive dependencies, meaning non-key attributes must only depend on the primary key.
  - Boyce-Codd Normal Form (BCNF): Strengthens 3NF by ensuring every determinant is a candidate key.
  - Higher Normal Forms (4NF, 5NF): Address rare cases of multi-valued dependencies and further refine table structures.
- Denormalization
  - The intentional introduction of redundancy to improve query performance, reducing the need for joins. It is useful in read-heavy systems, analytics, and cases where frequent complex queries impact speed.
- Trade-Offs of Denormalization
  - Pros: Faster read queries, fewer joins, better analytical performance.
  - Cons: Increased storage usage, risk of data inconsistency, more complex updates.

Lectures 15 - 17

Outline

- Introduction to Query Processing
- Steps in Query Processing
- Introduction to Query Execution Plans

Query Processing

- Definition:
  - Query processing is the overall procedure through which a high-level SQL query is translated into a series of low-level operations that the database can execute.
  - It involves parsing, translating, validating, optimizing, and executing queries to retrieve or modify data.
- Key Points:
  - Converts SQL queries into executable instructions.
  - Handles syntax and semantic checks.
  - Involves query rewriting and selecting an efficient execution strategy.

Query Optimization

- Definition:
  - Query optimization is a crucial phase within query processing that focuses on improving the performance of a query by selecting the most efficient execution plan.
  - The optimizer evaluates multiple possible query execution plans and chooses the one with the lowest estimated cost (in terms of time, memory, and other resources).
- Key Points:
  - Involves cost estimation for various execution strategies.
  - Uses heuristics, rules, and cost-based methods.
  - Significantly impacts query performance.

Query Execution Plan

- Definition:
  - A query execution plan (QEP) is the step-by-step roadmap generated by the database engine that shows how an SQL query will be executed.
  - It details the operations (e.g., joins, scans, sorts) and the order in which they will be performed.
- Key Points:
  - Generated after query optimization.
  - Can be viewed using commands like EXPLAIN in SQL.
  - Helps in understanding and improving query performance.

Comparison

Aspect	Query Processing	Query Optimization	Query Execution Plan
Focus	End-to-end handling of SQL queries.	Improving performance by selecting the best plan.	Blueprint of how the database will execute the query.
When It Occurs	Entire query lifecycle.	During the optimization phase of processing.	After optimization, before execution.
Purpose	Ensure correct and efficient execution.	Minimize resource usage and execution time.	Describe the exact operations for execution.
Output	Executable operations.	The most efficient execution plan.	Visual or textual representations of execution steps.

Example

- SELECT CustomerName FROM Customers WHERE City = 'Toronto';
- Query Processing:
  - Parses the SQL statement.
  - Checks syntax and semantics.
  - Converts it into relational algebra expressions.
- Query Optimization:
  - Considers multiple ways to retrieve data (e.g., full table scan vs. index search).
  - Chooses the most efficient method based on cost estimation.
- Query Execution Plan:
  - Shows the chosen steps, e.g.:
    - Use index on City column (if available).
    - Perform a filter operation (City = 'Toronto').
    - Return CustomerName values.

Why is Query Processing Essential in Databases?

1. Efficiency and Performance: It ensures that user queries are executed in the most efficient manner, reducing the time and resources required for data retrieval.
2. Optimization: Query processing uses optimization techniques to choose the best execution plan, leading to faster results even for complex queries.
3. Accurate Results: It translates high-level SQL queries into low-level operations, ensuring correct and consistent outputs from the database.
4. Resource Management: By optimizing queries, it reduces the load on the system, leading to better resource utilization, especially in multi-user environments.
5. Scalability: Efficient query processing allows databases to handle large datasets and high transaction volumes without compromising performance.
6. Security and Integrity: It enforces access controls and integrity constraints during query execution, maintaining data security and consistency.

What happens after a Query is Submitted?

- After a query is submitted to a Database Management System, it undergoes several stages involving query processing, query optimization, and the creation of a query execution plan.
- Query Processing
  - This is the initial phase where the DBMS prepares for the query for execution.
  - Parsing: The DBMS checks the SQL query for syntax errors and validates the semantics (e.g., verifying that tables and columns exist).
  - Translation: The valid SQL query is translated into an internal form, such as a relational algebra expression or logical plan.
  - Validation: Ensures that user permissions and integrity constraints and satisfied.
  - Query Rewriting: Sometimes, the DBMS rewrites the query into a logically equivalent but more efficient form (e.g., simplifying conditions or pushing filters).
- Query Optimization
  - The translated query may have multiple ways to execute.
  - The optimizer selects the most efficient execution plan based on cost estimation.
  - Enumerating Plans: The optimizer considers various strategies (e.g., join methods, access paths like index scans).
  - Cost Estimation: For each plan, it estimates resources like CPU time, I/O operations, and memory usage.
  - Choosing the Best Plan: The plan with the lowest estimated cost is selected for execution.
  - Example: If a query involves joining two tables, the optimizer decides whether to use a nested loop join, merge join, or hash join based on the data size and indexes.

Join Type	Best For	Complexity	Key Requirement	Notes
Nested Loop	Small datasets, indexed joins	$O(N \times M)$	Index helps performance	Slow for large joins
Merge Join	Large, sorted datasets	$O(N + M)$	Sorted join columns	Efficient if pre-sorted
Hash Join	Large, unsorted datasets	$O(N + M)$	Sufficient memory memory	Fast, but memory-heavy

- Query Execution Plan
  - This plan describes how the database will execute the query, including the exact order of operations.
  - Access Methods: Will it use a full table scan or an index scan?
  - Join Algorithms: What type of join will be used if there are multiple tables?
  - Operation Order: In what sequence will operations like filtering, joining, and sorting occur?
  - Physical Plan Details: The chosen plan is represented in a form that the DBMS execution engine can process.
- Query Execution
  - Finally, the DBMS follows the execution plan to run the query and produce the results.
  - The DBMS performs data retrieval, manipulation, and any required computations.
  - Results are returned to the user, and the DBMS may also cache the execution plan for future identical queries.

Difference between Logical and Physical Query Processing

- Logical Query Processing
  - Describes what the query intends to do.
  - It's DBMS-independent and focuses on relational operations needed to get the results.
  - Represents the high-level steps based on SQL semantics.
  - Deals with projections, selections, joins, grouping, and sorting without specifying how to execute them.
  - Shows the logical order of operations (which differs from SQL syntax order).
  - Example (SQL Query):
    - SELECT DepartmentName, COUNT(*) AS EmployeeCount FROM Employees JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID
      WHERE Salary >50000 GROUP BY DepartmentName;
  - Logical Processing Order:
    - FROM & Join - Combine Employees and Departments
    - WHERE - Filter rows where Salary >50000
    - GROUP BY - Group the filtered rows by DepartmentName
    - SELECT - Choose the DepartmentName and count employees
- Physical Query Processing
  - Describes how the DBMS actually executes the query.
  - It's DBMS-dependent, considering indexes, access paths, and join algorithms (like nested loop, merge join, hash join).
  - Focuses on execution strategies to optimize performance.
  - Involves query optimization and execution plans.
  - Decides between various methods, e.g., index scan vs. table scan.
  - Prioritizes efficiency, aiming for minimal I/O and CPU usage.
  - Example (SQL to View Execution Plan)

    - EXPLAIN ANALYZE SELECT DepartmentName, COUNT(*) AS EmployeeCount FROM Employees JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID WHERE Salary >50000 GROUP BY DepartmentName;
  - Physical Plan Example:
    - Use of a hash join for combining tables.
    - Index seek on Employees.Salary if an index exists.
    - Parallel execution if the DBMS supports it.

Using EXPLAIN

- EXPLAIN SELECT EmployeeName FROM Employees WHERE Department = 'IT';

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Employees	ref	idx_department	idx_department	20	const	10	Using where

- type: ref indicates an index scan using a non-unique index.
- key: idx_department is used (an index on Department).
- rows: Estimated number of rows (10) to check.
- Extra: Using where shows the filtering condition applied after the index scan.

Using EXPLAIN ANALYZE

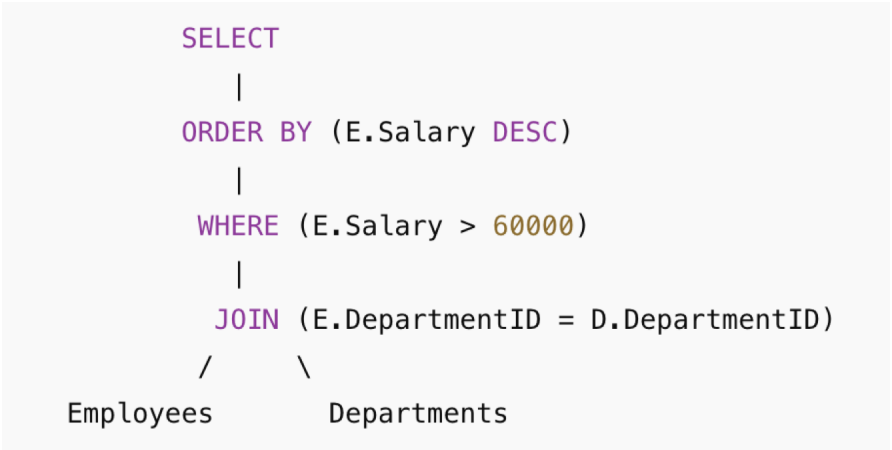
- EXPLAIN ANALYZE SELECT EmployeeName FROM Employees WHERE Department = 'IT';
- Index Scan using idx_department on employees (cost=0.15..8.37 rows=10 width=15) Index Cond: (department='IT'::text)
- Index Scan: PostgreSQL uses the index idx_department for faster lookup.
- Index Cond: The filtering condition applied directly during the index scan (department = 'IT').
- cost=0.15..8.37: Cost Estimated startup and total cost of executing the query.
- rows = 10: Estimated number of rows to be returned.
- width = 15: Average size (in bytes) of each row.

Steps in Query Processing

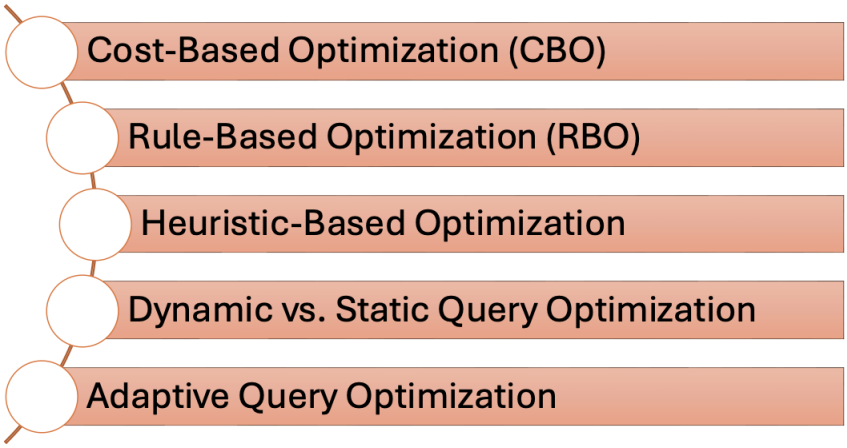
- Parsing and Translation
  - Parsing ensures the SQL query is syntactically and semantically correct.
  - Translation converts the query into an internal representation, typically a parse tree, that the DBMS can understand and optimize.
- Why Parsing & Translation Matter
  - Prevents execution errors: Early detection of issues saves computational resources.
  - Prepares for optimization: The parse tree serves as a blueprint for the query optimizer.
  - Improves performance: Efficient translation leads to faster query execution.
- Checking Syntax and Semantics

  - Syntax Checking:
    * Ensures that the SQL statement follows the correct grammatical structure.
    * Errors like missing keywords (SELECT, FROM), wrong punctuation, or invalid clause orders are caught here.
    * Example:
      · Incorrect Syntax (Missing FROM keyword) SELECT name, age Employees;
      · SELECT name, age Employees;
      · DBMS Response: Syntax error near 'Employees'; expected 'FROM'
  - Semantic Checking:
    * Verifies logical correctness of the query:
      · Are all columns and tables valid?
      · Do data types match for operations?
    * Example:
      · Semantic Error (Invalid column 'salaryy') SELECT name, salaryy FROM Employees;
      · SELECT name, salaryy FROM Employees;
      · DBMS Response: Column 'salaryy' does not exist in table 'Employees'

- Generating a Parse Tree
  - The parse tree represents the hierarchical structure of the SQL query.
  - Each node in the tree corresponds to an operation (e.g., SELECT, JOIN, WHERE).
  - The tree structure will guide the optimizer in planning how to execute the query efficiently.
  - Example:
    * SELECT D.DepartmentName, E.EmployeeName, E.Salary FROM Employees E JOIN Departments D ON E.DepartmentID = D.DepartmentID WHERE E.Salary >60000 ORDER BY E.Salary DESC;



- Query Optimization
  - Improves Performance: Faster query executions aves time and computing resources.
  - Scales with Data: Efficient queries handle larger datasets more effectively.
  - Reduces Costs: Less resource usage leads to lower operational costs.
- Approaches to Query Optimization



Approaches to Query Optimization

- Cost-Based Optimization (CBO)
  - The optimizer generates multiple execution plans and estimates the cost of each plan based on factors such as:
    - CPU usage
    - Disk I/O
    - Network latency
    - Memory usage
  - The plan with the lowest estimated cost is selected.
  - Key Features:
    - Uses statistics (e.g., table size, index availability, data distribution).
    - Considers query complexity, available indexes, and join methods.
    - Can choose between nested loop joins, hash joins, or merge joins based on costs.
  - Example: SELECT E.Name, D.DepartmentName FROM Employees E JOIN Departments D ON E.DeptID = D.DeptID WHERE E.Salary >100000;
  - CBO Decision:
    - If an index exists on Salary, it will use it to avoid a full table scan.
    - If Employees is large and Departments is small, a hash join might be selected.
  - Examples of Nested Loop Join, Hash Join, and Merge Join with Cost Calculation
  - Employees

EmpID	Name	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10

  - Departments

DeptID	DeptName
10	HR
20	IT
30	Marketing

  - SELECT E.Name, D.DeptName FROM Employees E JOIN Departments D ON E.DeptID = D.DeptID;
  - Nested Loop Join

    * For each row in the outer table (Employees), scan all rows in the inner table (Departments) to find matching rows.
    * Best for: Small tables or when indexes exist on the join column.
    * Cost Calculation:
        · Cost Formula: Cost $= T1 \times T2$
        · $T1$ = Number of rows in outer table (Employees)
        · $T2$ = Number of rows in inner table (Departments)
        · Cost: $3 \times 3 = 9$ comparisons.
    * Example Process:
    * Step 1: Take Alice (DeptID = 10) from Employees. Search Departments for DeptID = 10.
    * Step 2: Take Bob (DeptID = 20). Search Departments for DeptID = 20.
    * Step 3: Take Charlie (DeptID = 10). Search Departments for DeptID = 10.
- Hash Join
    * Build Phase: Create an in-memory hash table from the smaller table (Departments) using the join key (DeptID).
    * Probe Phase: For each row in Employees, hash the join key and look up the match in the hash table.
    * Best for: Large datasets without indexes.
    * Cost Calculation:
        · Cost Formula: Cost $= B(T2) + P(T1)$
        · $B(T2)$: Build hash cost (proportional to smaller table size)
        · $P(T1)$: Probe cost (proportional to outer table size)
        · Cost $= 3 + 3 = 6$ comparisons.
    * Example Process:
    * Build Hash Table:
        · Key: DeptID $\rightarrow$ Value: DeptName
        · $\{10 : \text{'HR}', 20 : \text{'IT}', 30 : \text{'Marketing}'\}$
    * Probe:
        · Alice (DeptID = 10) $\rightarrow$ Hash lookup $\rightarrow$ HR
        · Bob (DeptID = 20) $\rightarrow$ Hash lookup $\rightarrow$ IT
        · Charlie (DeptID = 10) $\rightarrow$ Hash lookup $\rightarrow$ HR
- Merge Join
    * Both tables must be sorted on the join key (DeptID).
    * The algorithm scans both tables simultaneously, merging matching rows.
    * Best for:
        · Pre-sorted data
        · Large datasets with sorted indexes
    * Cost Calculation:
        · Cost Formula: Cost $= S(T1) + S(T2) + (T1 + T2)$
        · $S(T1)$ and $S(T2)$: Sorting costs for each table.
        · $T1 + T2$: Merge scan cost.
        · Cost: $0 + 0 + (3 + 3) = 6$ comparisons.
    * Example Process:
    * Step 1: Sort both tables on DeptID (if not already sorted)
    * Step 2:
        · Compare the first rows of each table.
        · Advance in the table with the lower key until a match is found.

        * Example matches:
            · (Alice, HR), (Charlie, HR) for DeptID = 10
            · (Bob, IT) for DeptID = 20
    - Comparison of Join Types:

Join Type	Best For	Cost Complexity	Advantages	Disadvantages
Nested Loop	Small datasets indexed columns	$O(T1 \times T2)$	Simple, good with indexes	Slow on large datasets
Hash Join	Large, unsorted datasets	$O(T1 + T2)$	Fast with no sorting required	Requires extra memory for hash
Merge Join	Large, sorted datasets	$O(T1 + T2) + \text{sort}$	Efficient on pre-sorted data	Sorting overhead if unsorted

- Rule-Based Optimization (RBO)
    - Uses predefined rules instead of cost estimates.
    - Examples of rules:
        * Use indexes if available.
        * Perform selection before join.
        * Join small tables first.
    - Example: SELECT E.Name, D.DepartmentName FROM Employees E JOIN Departments D ON E.DeptID = D.DeptID WHERE E.Salary >100000;
    - For the above query, RBO might:
        * Perform Salary filter first.
        * Then, join filtered Employees with Departments.
- Heuristic-Based Optimization
    - Applies heuristics (rules of thumb) to transform a query into an efficient form.
    - These heuristics are general guidelines rather than cost-based decisions.
    - Common heuristics:
        * Perform selection operations early (reduce dataset size before joins).
        * Perform projection operations early (reduce the number of columns).
        * Replace Cartesian products with joins when possible.
        * Reorder joins to start with relations producing fewer tuples.
    - Example Heuristic Application:
        * SELECT E.Name, D.DepartmentName FROM Employees E JOIN Departments D ON E.DeptID = D.DeptID WHERE E.Salary >100000;
        * Step 1: Filter Employees (Salary >100000).
        * Step 2: Join filtered Employees with Departments.
        * Step 3: Select only Name and DepartmentName.
- Static Optimization
    - The execution plan is generated once when the query is compiled.
    - No re-optimization occurs, regardless of data changes.
    - Advantage: Faster execution after compilation.
    - Used in: Stable environments where data distribution is consistent.
- Dynamic Optimization
    - Optimization occurs at runtime.

  - The execution plan is generated just before execution, considering current data distribution.
  - The query optimizer uses the most up-to-date statistics and index information available at runtime.
  - Advantage: Can adapt to real-time changes in data.
  - Used in: Systems where data changes frequently.

- Adaptive Query Optimization
  - Adaptive query optimization takes optimization a step further by continuously monitoring the query execution and adapting the plan during execution.
  - Occurs during execution, not just before.
  - Uses real-time feedback from partial execution results to adjust:
    * Join methods (e.g., switch from nested loop to hash join)
    * Join orders
    * Parallelism levels
  - Common in modern databases like Oracle 12c+, SQL Server, and PostgreSQL with adaptive capabilities.
  - Helps with data skew and unpredictable data distributions.

Index Usage

- Indexes speed up data retrieval by avoiding full table scans.
- Example with Index:
  - CREATE INDEX idx_salary ON Employees(Salary);
  - Now, the optimizer can:
    * Quickly locate employees with Salary >80000.
    * Reduce I/O costs by skipping irrelevant rows.

Join Order Importance

- The order in which tables are joined affects performance.
- Joining smaller tables first often reduces the size of intermediate results.
- Example: SELECT E.EmployeeName, P.ProjectName FROM Employees E JOIN WorksOn W ON E.EmployeeID = W.EmployeeI
  JOIN Projects P ON W.ProjectID = P.ProjectID WHERE E.DepartmentID = 3;
- Optimized Join Order:
  - Filter Employees by DepartmentID.
  - Join the filtered result with WorksOn.
  - Finally, join with Projects.

What is a Query Execution Plan?

- A Query Execution Plan (QEP) is a detailed roadmap that a Database Management System (DBMS) uses to execute an SQL query.
- It shows the sequence of operations (like table scans, joins, sorts, and aggregations) that the DBMS will perform to retrieve the requested data as efficiently as possible.
- The query execution engine is the component of a Database Management System responsible for executing the steps outlined in the query execution plan.
- It transforms the high-level SQL query into low-level operations that interact with the database's data storage.

Components of a Query Execution Plan

- Access Methods:
  - Table Scan: Reads every row in a table.
  - Index Scan / SEek: Uses indexes for faster lookups.
- Join Methods:
  - Nested Loop Join
  - Hash Join
  - Merge Join
- Sorting and Aggregation:
  - ORDER BY, GROUP BY operations.
- Filter Conditions:
  - Conditions like WHERE clauses applied during execution.
- Projection:
  - Projection refers to selecting specific columns from a table rather than retrieving all columns (SELECT *).
  - It reduces the amount of data processed and transferred, improving query performance.
- Combining Results (Set Operations):
  - Set operations combine results from multiple queries:
  - UNION: Combines distinct rows from two queries.
  - UNION ALL: Combines all rows (including duplicates).
  - INTERSECT: Returns rows common to both queries.
  - EXCEPT (MINUS): Returns rows from the first query not in the second.
- Cost Estimates:
  - Shows how resource-intensive each step is, helping prioritize optimizations.

Example of a Complex Query and It's Execution Plan

- Tables:
  - Employees
    - EmployeeID, Name, DepartmentID, Salary
  - Departments
    - DepartmentID, DepartmentName, ManagerID
  - Projects
    - ProjectID, ProjectName, DepartmentID
  - EmployeeProjects
    - EmployeeId, ProjectID, HoursWorked
- Complex Query:
  - SELECT e.Name, d.DepartmentName, p.ProjectName, SUM(ep.HoursWorked) AS TotalHours FROM Employees e JOIN Departments d ON e.DepartmentID = d.DepartmentID JOIN EmployeeProjects ep ON e.EmployeeID = ep.EmployeeID JOIN Projects p ON ep.ProjectID = p.ProjectID WHERE e.Salary >60000 GROUP BY e.Name, d.DepartmentName, p.ProjectName HAVING SUM(ep.HoursWorked) >100 ORDER BY TotalHours DESC;
- Execution Plan Breakdown:

Step	Operation	Object	Description	Cost Impact
1	Index Seek	Employees (SalaryIDX)	Efficiently retrieves employees with Salary >60000.	Low
2	Nested Loop Join	Employees, Departments	For each employee, find matching department.	Medium
3	Hash Join	EmployeeProjects	Joins large sets using hashing for performance.	High
4	Merge Join	Projects	Efficient join on ProjectID with sorted data.	Medium
5	Group By (Aggregation)	-	Groups data by Name, DepartmentName, ProjectName. Calculates SUM(HoursWorked).	High
6	Filter (HAVING)	-	Keeps groups where SUM(HoursWorked) >100.	Medium
7	Sort (Order By)	-	Sorts the final result by TotalHours DESC.	High

- To optimize the given complex query execution plan, we can focus on reducing I/O, improving join efficiency, and minimizing costly operations like sorting and grouping.
  - Optimize Indexing for Faster Lookups
    - Current Issue: The GROUP BY, HAVING, and JOIN operations are costly.
    - Solution: Create covering indexes that support filtering, joining, and grouping columns.
    - CREATE INDEX idx_employee_salary ON Employees(Salary);
    - CREATE INDEX idx_employee_department ON Employees(DepartmentID);
    - CREATE INDEX idx_ep_project ON EmployeeProjects(ProjectID, EmployeeID, HoursWorked);
    - CREATE INDEX idx_project_department ON Projects(DepartmentID);
  - Reorder Joins Based on Selectivity
    - Current Issue: The join order affects performance - high cardinality joins first slow down execution.
    - Solution: Join filtered tables first to reduce intermediate results.
    - Original Join Order: Employees → Departments → EmployeeProjects → Projects
    - Optimized Join Order: Employees (filtered by Salary) → EmployeeProjects → Projects → Departments
  - Do Aggregations Early
    - Current Issue: Aggregations are performed after joining large datasets.
    - Solution: Pre-aggregate HoursWorked before joining with other tables:

    - WITH ProjectHours AS (SELECT EmployeeID, ProjectID, SUM(HoursWorked) AS TotalHours FROM EmployeeProjects
      GROUP BY EmployeeID, ProjectID HAVING SUM(HoursWorked) >100)
    - SELECT e.Name, d.DepartmentName, p.ProjectName, ph.TotalHours FROM Employees e JOIN ProjectHours pj ON e.EmployeeID = ph.EmployeeID
      JOIN Projects p ON ph.ProjectID = p.ProjectID JOIN Departments d ON e.DepartmentID = d.DepartmentID WHERE e.Salary >60000 ORDER BY ph.TotalHours DESC;
  - Reduce Sorting Overhead
    - Current Issue: The ORDER BY clause triggers a costly sort.
    - Solution: Use indexes that align with the ORDER BY clause or window functions:
    - CREATE INDEX idx_hours_order ON EmployeeProjects(HoursWorked DESC);

Summary

- Query Processing
  - Translates SQL queries into low-level database operations.
  - Steps include parsing, validating, optimizing, and executing queries.
- Query Optimization
  - Selects the most efficient execution plan to minimize cost (CPU, memory, I/O).
  - Uses heuristics, rule-based, and cost-based optimization methods.
- Query Execution Plan (QEP)
  - The roadmap for executing an SQL query.
  - Includes operations like joins, scans, and sorting.
  - Can be analyzed using EXPLAIN or EXPLAIN ANALYZE.
- Steps in Query Processing
  - Parsing & Translation - Checks syntax and converts query into internal representation.
  - Query Optimization - Evaluates different execution strategies and selects the best one.
  - Query Execution Plan - Details the exact operations for execution.
  - Query Execution - The database follows the plan to fetch and return results.
- Logical vs. Physical Query Processing
  - Logical: Describes what the query does (relational operations).
  - Physical: Describes how the DBMS executes it (access paths, join methods).
- Join Methods
  - Nested Loop Join: Good for small datasets but slow for large ones.
  - Merge Join: Efficient for pre-sorted datasets.
  - Hash Join: Fast for large datasets but memory-intensive.
- Query Optimization Approaches
  - Cost-Based (CBO): Chooses the lowest-cost execution plan.
  - Rule-Based (RBO): Uses predefined rules.
  - Heuristic-Based: Applies general optimization rules.

  - Static vs. Dynamic: Static plans don't change, while dynamic plans adapt at runtime.
  - Adaptive Optimization: Adjusts execution plans during execution based on real-time feedback.
- Indexing & Join Order
  - Indexes speed up query execution by avoiding full table scans.
  - Optimized join orders reduce intermediate data size and improve performance.

Lecture 18

Outline

- Transaction Concept & ACID Properties
- SImple Transaction Model
- Storage Structure
- Logging for Atomicity and Durability
- Transaction Isolation
- Serializability

Transaction Concept & ACID Properties

- Definition of a Transaction
  - A transaction is a unit of execution that accesses and updates data.
  - Initiated via SQL or programming langues (e.g., C++, Java, JDBC, ODBC)
  - Bounded by BEGIN TRANSACTION and END TRANSACTION statements.

```
BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 500 WHERE account_id = 1;

UPDATE Accounts SET balance = balance + 500 WHERE account_id = 2;

-- check if both operations were successful
IF @ERROR = 0
BEGIN
    COMMIT;
    END TRANSACTION;
END
ELSE
BEGIN
    ROLLBACK;
    END TRANSACTION;
END
```

- A database system is required to maintain the following properties of transactions:
  1. **Atomicity (All-or-Nothing)**
     - A transaction must be fully completed or fully undone if it fails.

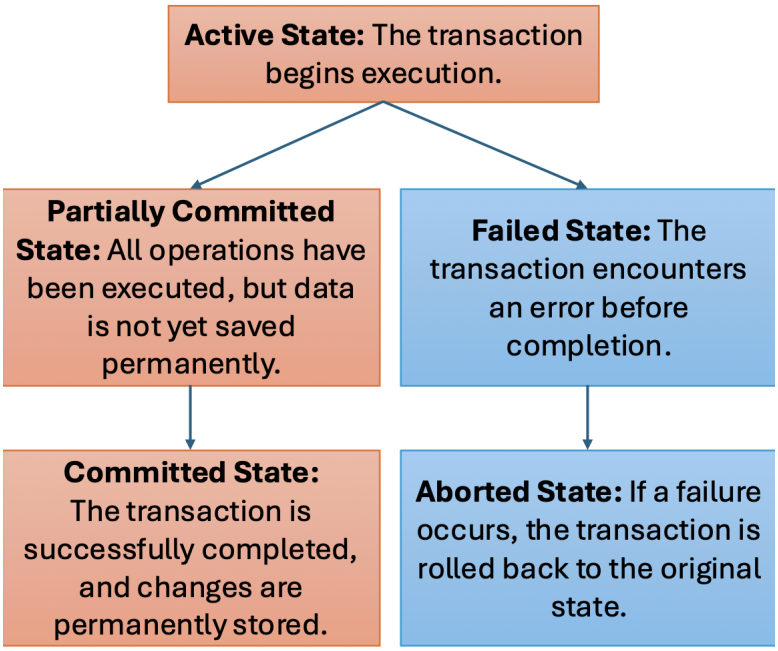
- If a system crashes or an error occurs, any partial changes must be rolled back.

2. **Consistency (Maintaining Integrity)**
   - Ensures the database remains in a valid state before and after a transaction.
   - Transactions must enforce constraints like primary keys, foreign keys, and business rules.
3. **Isolation (Concurrency Control)**
   - Each transaction executes as if it were the only one running.
   - Prevents interference between concurrent transactions.
4. **Durability (Permanent Changes)**
   - Once a transaction commits, its changes must persist even after system failures.

- ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**.
- Strict enforcement can impact performance.
- Some applications relax isolation for better speed (e.g., banking systems, online booking).
- **Atomicity (All-or-Nothing)**
  - Example: Online Banking Transfer
  - Suppose Alice transfers $500 from her account to Bob's. the transaction consists of two operations:
    * Deduct $500 from Alice's account.
    * Add $500 to Bob's account.
  - If the system crashes after Step 1 but before Step 2, Alice loses $500, but Bob doesn't receive it.
  - Atomicity ensures that both steps complete or non occur, so the system rolls back any partial changes.
- **Consistency (Maintaining Integrity)**
  - A customer places an order, which requires:
    * Deducting stock from the inventory.
    * Creating an order record in the database.
  - If stock is not updated correctly while the order is created, the database becomes inconsistent (selling items not in stock).
  - Consistency ensures that only valid state transitions occur, so either both steps complete, or none occur.
- **Isolation (Concurrency Control)**
  - Example: Flight Seat Reservation
  - Two users, A and B, try to book the last seat on a flight at the same time.
  - Without isolation, both may see the seat as available and both may be charged, leading to overbooking.
  - Isolation ensures that transactions execute as if they are alone, preventing data conflicts.
- **Durability (Permanent Changes)**
  - Example: ATM Withdrawal
  - A user withdraws $200 from an ATM, and the transaction successfully commits.
  - If a power failure occurs immediately after, the system must not lose track of the withdrawal.
  - Durability ensures the transaction is permanently saved, even after a system crash.

Simple Transaction Model

- Transaction Basics:
  - A transaction is a sequence of operations that access and modify data.
  - Basic operations: read(X) (fetches data into memory) and write(X) (writes data back to disk).
  - Ensuring whether changes are in memory or committed to disk is crucial.
- Example Transaction - Transferring $50 from A to B:
  - read(A); A := A - 50; write(A);
  - read(B); B := B = 50; write(B);
  - Ensures proper balance transfer between accounts.



Transaction Lifecycle

Storage Structure

- Storage Types:
  - Volatile Storage:
    - Fast storage (e.g., main memory (cache)).
    - Data is lost on system crashes.
  - Non-Volatile Storage:
    - Survives system crashes (e.g., hard disks, SSDs, optical media).
    - Slower than volatile storage and can still fail.
  - Storage Structure:
    - Designed to ensure no data loss (theoretically).
    - Achieved by replicating data on multiple non-volatile storage devices.
- Challenges in Ensuring Durability and Atomicity:

  - Durability: A transaction's changes must be written to stable storage to persist.
  - Atomicity: Log records must be stored in stable storage before database modifications.
- Practical Considerations:
  - Some systems use battery-backed memory to retain volatile storage after crashes.
  - RAID and multiple disk copies improve stability and durability.
  - Systems with high-value data require stronger durability measures.

Logging for Atomicity and Durability

- A log records:
  - Transaction ID
  - Data item ID
  - Old and new values of modified data items
- Logging enables:
  - Undoing modifications (atomicity)
  - Redoing modifications (durability)
- Transaction Commitment
  - A transaction that completes successfully is committed.
  - A committed transaction's updates persist even after system failure.
  - Compensating transactions may be needed to reverse committed changes but are not always possible.
- Handling Failures
  - After failure, the system can either:
    * Restart the transaction (if failure was external).
    * Terminate the transaction (if failure was due to internal logic errors).
  - Failures can result from hardware, software, or logical errors.

Transaction Isolation

- Concurrency in Transactions
  - Multiple transactions can run concurrently, improving system efficiency.
  - Running transactions serially ensures consistency but is inefficient.
- Advantages of Concurrency
  - Improved throughput and resource utilization:
    * CPU and disk operations can be performed in parallel.
    * Reduces idle time of system resources.
  - Reduced waiting time:
    * Short transactions don't have to wait for long ones.
    * Minimizes delays and improves response time.
- Challenges of Concurrency

  - Isolation property may be violated, leading to inconsistent database states.
  - Transactions must be controlled to maintain database consistency.
- Schedules in Transactions
  - Serial Schedule: Transactions execute one after another in sequence.
  - Concurrent Schedule: Transactions are interleaved but must maintain consistency.
- Example Transactions:
  - T1: Transfers $50 from Account A to B.
  - T2: Transfers 10% of Account A's balance to Account B.
  - Serial execution: Ensures correctness and consistency.
  - Concurrent execution: Can be correct if properly controlled but may lead to inconsistencies if not managed.

Table 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Table 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Table 17.5 Schedule 4—a concurrent schedule resulting in an inconsistent state.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

- Serializable Schedules
  - A concurrent schedule is correct if its final result matches some serial execution.
  - The database system ensures only serializable schedules are executed to maintain consistency.
- Concurrency Control
  - Left unmanaged, concurrent execution may cause inconsistencies.
  - Concurrency control mechanisms ensure correct execution by enforcing serializability.

Serializability

- Definition and Importance
  - Serializability ensures that concurrent execution of transactions produces the same results as some serial execution.
  - Serial schedules are always serializable, but determining serializability for interleaved schedules is more complex.
- Read and Write Operations
  - Transactions are considered based on two primary operations: read(Q) and write(Q).
- Conflicting Instructions
  - Two instructions conflict if they involve the same data item and at least one is a write operation.
  - Conflict cases:
    * read(Q) - read(Q): No conflict, order does not matter.

    - read(Q) - write(Q): Order matters.
    - write(Q) - read(Q): Order matters.
    - write(Q) - write(Q): Order affects the final database state.

- Conflict Serializability
  - A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
- Testing Conflict Serializability
  - Construct the precedence graph based on read / write dependencies.
  - Check for cycles: If a cycle exists, the schedule is not serializable.
  - Topological Sorting: If the graph is acyclic, transactions can be ordered to form a serial schedule.
- Precedence Graph (Dependency Graph)
  - A direct graph (V,E) is constructed to check serializability.
  - Nodes (V): Represent transactions.
  - Edges (E): Exist if one transaction's read / write depends on another's write operation.
  - Cycle Detection: If the graph contains a cycle, the schedule is not conflict serializable.
- Example 1: Conflict Serializable Schedule
  - Consider the following schedule involving transactions T_1 and T_2:

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

  - Step 1: Identify Conflicting Instructions
    - write(A) (T_1) → read(A) (T_2) → Conflict (write-read on A)
    - write(A) (T_1) → write(A) (T_2) → Conflict (write-write on A)
    - write(B) (T_1) → read(B) (T_2) → Conflict (write-read on B)
    - write(B) (T_1) → write(B) (T_2) → Conflict (write-write on B)
  - Step 2: Construct Precedence Graph
    - $T_1 \rightarrow T_2$ because T_1 writes A before T_2 reads / writes it.
    - $T_1 \rightarrow T_2$ because T_1 writes B before T_2 reads / writes it.
  - Since there is no cycle, the schedule is conflict serializable. It is equivalent to the serial execution $T_1 \rightarrow T_2$.
- Example 2: Non-Conflict Serializable Schedule
  - Consider the following schedule with T_3 and T_4:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

  - Step 1: Identify Conflicting Instructions
    - read(Q) (T_3) → write(Q) (T_4) → Conflict (read-write on Q)
    - write(Q) (T_4) → write(Q) (T_3) → Conflict (write-write on Q)

  - Step 2: Construct Precedence Graph
    - $T_3 \rightarrow T_4$ because T_3 reads Q before T_4 writes it.
    - $T_4 \rightarrow T_3$ because T_4 writes Q before T_3 writes it.
  - Step 3: Check for Cycles
    - The precedence graph contains a cycle ($T_3 \rightarrow T_4 \rightarrow T_3$).
    - Since there is a cycle, the schedule is not conflict serializable.
  - This means the schedule cannot be transformed into an equivalent serial execution.

- Limitations of Conflict Serializability
  - Some schedules may produce the same final result as a serial schedule but are not conflict serializable.
  - Example: Commutative operations like incrementing / decrementing values may lead to identical final states despite conflicting order.
  - View Serializability exists but is computationally expensive and rarely used in practice.

Summary

- Transaction Concept & ACID Properties
  - A transaction is a unit of execution that modifies data, initiated via SQL or programming languages.
  - Transactions maintain ACID properties:
    - Atomicity: All operations must complete successfully or be rolled back.
    - Consistency: Database remains valid before and after a transaction.
    - Isolation: Transactions execute independently to prevent conflicts.
    - Durability: Committed changes persist even after system failures.

- Simple Transaction Model
  - A transaction consists of read(X) and write(X) operations.
  - Example: Transferring $50 from Account A to B involves:
    - Reading and deducting from A $\rightarrow$ Writing to A
    - Reading and adding to B $\rightarrow$ Writing to B

- Storage Structure
  - Volatile storage (e.g., RAM) loses data on crashes.
  - Non-volatile storage (e.g., HDD, SSD) preserves data but is slower.
  - Systems use RAID, backups, and logging to ensure durability.

- Logging for Atomicity & Durability
  - Logs track transaction details (ID, old/new values).
  - Enables rollback (undo) for atomicity and recovery (redo) for durability.
  - After failures, transactions can be retried or aborted based on error type.

- Transaction Isolation & Concurrency Control
  - Concurrency improves efficiency but risks inconsistent data.
  - Schedules:
    - Serial Schedule: Transactions execute sequentially $\rightarrow$ always consistent.

    - Concurrent Schedule: Transaction interleave → requires control for consistency.
  - Serializable Schedules: Ensure the same result as a serial execution.

- Serializability
  - Conflict Serializability: Transaction must follow a serial order by analyzing read/write conflicts.
  - Precedence Graph helps detect cycles:
    - No cycle → Conflict serializable.
    - Cycle exists → Not conflict serializable.
  - Limitations: Some schedules may be view serializable (producing the same result as a serial schedule) but not conflict serializable.

Lecture 19

Outline

- Transaction Isolation and Atomicity
- Transaction Isolation Levels

Transaction Isolation and Atomicity

- Transaction Failures and Atomicity
  - In concurrent execution, transaction failures must be handled to maintain atomicity.
  - If a transaction T_i fails, any transaction T_j that depends on T_i (i.e., has read its written data) must also be aborted.
  - To ensure atomicity, certain schedule restrictions are necessary.
- Recoverable Schedules
  - A recoverable schedule ensures that if T_j reads data written by T_i, then T_i must commit before T_j commits.
  - Example: If T_6 writes A and T_7 reads A, T_7 must commit after T_6 commits.
  - If T_j commits before T_i and T_i fails, recovery becomes impossible → Nonrecoverable Schedule.
  - Condition for recoverability: Ensure that dependent transactions commit in order

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- In this example, for the schedule to be recoverable, T7 would have to delay committing until after T6 commits.
- Cascadeless Schedules
  - Cascading rollback happens when a transaction failure leads to a chain of rollbacks due to dependencies.
  - Example: T_8 writes A, read by T_9, which writes A, read by T_10. If T_8 fails, all dependent transactions (T_9, T_10) must be rolled back.
  - Cascading rollbacks are inefficient and undesirable.
  - Cascadeless schedules prevent cascading rollbacks by ensuring that a transaction reads data only after the writer transaction commits.

– Every cascadeless schedule is also recoverable.

T_8	T_9	T_10
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Transaction Isolation Levels

- Serializability ensures database consistency by allowing transactions to execute as if they were executed sequentially.
- However, enforcing strict serializability can limit concurrency and affect performance.
- Some applications use weaker isolation levels to improve concurrency, placing additional responsibility on programmers to maintain correctness.
- SQL Isolation Levels (Ordered from Strongest to Weakest)

1. Serializable:
   - Ensures serializable execution, though some database implementations may allow non-serializable executions in certain cases.
2. Repeatable Read:
   - Prevents uncommitted reads and ensures that once a transaction reads a value, it cannot be modified by another transaction until it finishes.
   - However, transactions may still be non-serializable when searching for data.
3. Read Committed:
   - Allows transactions to read only committed data but does not prevent non-repeatable reads (i.e., a value read twice may change due to another transaction's commit).
4. Read Uncommitted:
   - Allows reading uncommitted data from other transactions.
   - Lowest isolation level, leading to potential inconsistencies.

- Dirty writes (writing to a data item modified by an uncommitted transaction) are disallowed at all levels.
- Default isolation level in most databases: Read Committed.
- Setting Transaction Isolation Levels
  - SQL allows setting the isolation level explicitly using:
  - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
- Different databases have specific syntax:
  - Oracle: ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
- Performance vs. Consistency Trade-Off

- Ensuring serializability may cause transactions to wait or abort if conflicts arise.
- Weaker isolation levels improve performance but increase the risk of inconsistencies.
- Some databases implement weaker isolation levels even when serializable is set, requiring application designers to handle inconsistencies.

Summary

- Transaction Isolation and Atomicity
  - Transaction Failures & Atomicity:
    - If Ti fails, any transaction Tj that reads Ti's data must also abort to maintain atomicity.
    - Restrictions ensure transactions commit in a valid order.
  - Recoverable Schedules:
    - A schedule is recoverable if a transaction commits only after all transactions it depends on have committed.
    - If Tj reads from Ti, Tj must commit after Ti to prevent inconsistency.
    - Nonrecoverable schedules occur when a dependent transaction commits before its predecessor.
  - Cascadeless Schedules:
    - Cascading rollbacks occurs when a failure leads to a chain of rollbacks.
    - Cascadeless schedules prevent cascading rollbacks by ensuring transactions only read committed data.
    - Every cascadeless schedule is recoverable.
- Transaction Isolation Levels
  - Serializability: Ensures transactions execute in a serial order but can limit performance.
  - SQL Isolation Levels (from Stronger to Weakest):
    - Serializable: Transactions execute in a serializable order.
    - Repeatable Read: Prevents uncommitted reads, but searches may still be non-serializable.
    - Read Committed (default in most databases): Allows only committed reads but does not prevent non-repeatable reads.
    - Read Uncommitted: Allows reading uncommitted data, leading to inconsistencies.
  - Setting Isolation Levels:
    - SQL command: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    - Oracle: ALTER SESSION SET ISOLATION LEVEL = SERIALIZABLE;
  - Performance vs. Consistency Trade-Off:
    - Stronger isolation ensures consistency but reduces concurrency.
    - Weaker isolation improves performance but requires developers to handle inconsistencies manually.

Lecture 20

Outline

- Introduction to Concurrency Control
- Concurrency Control Techniques
- Locking Mechanisms in DBMS

Introduction to Concurrency Control

- Concurrency Control in a database system refers to the set of techniques used to manage simultaneous operations without causing conflicts, inconsistencies, or data corruption.
- It ensures that transactions execute safely in a multi-user environment while preserving ACID (Atomicity, Concurrency, Isolation, Durability) properties.
- Why is Concurrency Control Needed?
  - In a multi-user database system, multiple transactions execute concurrently.
  - Concurrency can lead to issues like:
    - Lost Updates: When two transactions update the same data simultaneously, leading to inconsistent data.
    - Dirty Reads: A transaction reads uncommitted changes of another transaction.
    - Non-Repeatable Reads: Different values are read at different times due to other transactions.
    - Phantom Reads: A transaction sees new rows inserted by another transaction.
- Concurrency Control Goals
  - Maintain database consistency.
  - Ensure ACID properties (Atomicity, Consistency, Isolation, Durability).
  - Maximize transaction throughput and minimize waiting time.
- Example: Lost Updates
  - Problem: Two transactions update the same data item simultaneously, causing one update to be lost.
  - Consider a bank account balance table:

account_id	balance
101	500

  - Transaction T1: Reads balance = 500, adds + 100, and updates the balance to 600.
  - Transaction T2: Reads balance = 500, subtracts - 50, and updates the balance to 450.
  - Step-by-Step Execution (without concurrency control)

    - T1 reads balance = 500
    - T2 reads balance = 500
    - T1 adds 100 → new balance = 600 (not yet committed)
    - T2 subtracts 50 → new balance = 450
    - T1 writes 600 to the database
    - T2 writes 450 to the databases (overwrites T1's update)
  - Expected Balance: 550 (500 + 100 - 50)
  - Actual Balance: 450 (Update from T1 is lost!)

- Example: Dirty Reads
  - Problem: A transaction reads data that another transaction has modified but not committed, leading to inconsistency if the uncommitted changes are later rolled back.

account_id	balance
102	500

  - T1: Updates the balance to 600 but has not yet committed.
  - T2: Reads the balance as 600, assuming it's correct.
  - T1: Later rolls back the update, restoring balance to 500.
  - Step-by-Step Execution (without concurrency control)
    - T1 updates balance from 500 → 600 (uncommitted change)
    - T2 reads balance = 600 (uncommitted)
    - T1 rolls back (reverts balance = 500)
    - T2 now has incorrect information (balance = 600 was never real)
  - Incorrect Data: T2 made decisions based on an invalid balance.

- Example: Non-Repeatable Reads
  - Problem: A transaction reads the same data multiple times but gets different values because another transaction modified the data in between.

account_id	balance
103	1000

  - T1: Reads balance = 1000 at time T = 1.
  - T2: Updates balance to 1200 and commits.
  - T1: Reads the same balance again at time T = 2, but now it's 1200.
  - Step-by-Step Execution (without concurrency control)
    - T1 reads balance = 1000
    - T2 updates balance = 1200 and commits
    - T1 reads balance again → now 1200 (unexpected change)
  - Inconsistency: T1 expected the balance to remain 1000, but it changed midway.

- Example: Phantom Reads
  - Problem: A transaction retrieves a set of rows but finds new rows inserted by another transaction when it re-executes the query.

employee_id	name	department
201	Alice	Sales
202	Bob	Sales

  - A manager checks the list of employees in a department.

- T1: Queries SELECT * FROM employees WHERE department = 'Sales' → Gets Alice & Bob.
- T2: Inserts Charlie into the Sales department.
- T1: Reruns the same query → Now sees Alice, Bob, and Charlie.
- Step-by-Step Execution (without concurrency control)
  * T1 selects employees in Sales → Gets Alice & Bob
  * T2 inserts Charlie into Sales and commits
  * T1 selects employees again → Now sees Alice, Bob, and Charlie
- Unexpected Behaviour: T1 expected the same result, but a new row appeared.

Concurrency Control Techniques

- To avoid these issues, concurrency control mechanisms are used. The main approaches include:

1. Lock-Based Concurrency Control
   - Uses shared (S) and exclusive (X) locks to regulate access to data.
   - Two-Phase Locking (2PL) ensures serializability but can lead to deadlocks.
2. Timestamp-Based Concurrency Control
   - Assigns timestamps to transactions and enforces execution based on timestamps.
   - Prevents conflicts but may lead to frequent transaction restarts.
3. Optimistic Concurrency Control (OCC)
   - Transactions execute without restrictions and validate before committing.
   - Works well when conflicts are rare but can result in high rollback rates in high-contention environments.

Locking Mechanisms in DBMS

- Locking mechanisms in a Database Management System (DBMS) are concurrency control techniques used to manage access to data by multiple transactions.
- They prevent conflicts, data inconsistency, and concurrency-related issues by restricting access to data items through locks.
- A lock is a flag placed on a data item that determines whether other transactions can read or modify it.
- Types of Locks:
  - Binary Locks
    * A simple lock with two states: Locked (1) or Unlocked (0).
    * If locked, no other transaction can access the data.
    * If unlocked, any transaction can access it.
  - Shared (S) and Exclusive (X) Locks
    * Shared Lock (S): Allows multiple transactions to read a data item but not modify it.
    * Exclusive Lock (X): Allows only one transaction to read and modify the data item.
- Binary Locks Example

- Scenario: A bank account balance is stored in a table.

account_id	balance
101	500

- Two transactions (T1 and T2) try to access it simultaneously.
- Advantage: Simple to implement.
- Disadvantage: Prevents even read operations from running in parallel.

Step-by-Step Execution Table (Binary Locking)

Step	T1 Operations	T2 Operations	Lock Status on Balance (Account 101)
1	Requests to read Balance = 500	-	Locked (1) (T1 acquires the lock)
2	Reading Balance = 500	Requests to read Balance → ❌ **Blocked**	Locked (1) (T1 still holds the lock)
3	Updates Balance = 600	-	Locked (1) (T1 still holds the lock)
4	Commits the update and **releases lock**	-	Unlocked (0) (Lock is now free)
5	-	Reads Balance = 600 ✅	Locked (1) (T2 acquires lock temporarily)
6	-	Commits and releases lock	Unlocked (0)

- Shared (S) and Exclusive (X) Locks Example
  - Shared (S) Lock: Multiple transactions can read the data but cannot write.
  - Exclusive (X) Lock: Only one transaction can read and write at a time.
  - Scenario: A library database stores the number of copies of a book.
  - Step-by-Step Execution (S and X Locks)
  - Step 1: Two transactions Try to Read the Book Count
    - T1 applies for a Shared (S) lock on book_id = 201 → Granted
    - T2 also applies for a Shared (S) lock on book_id = 201 → Granted (Both transactions can read)
  - Step 2: A Transaction Wants to Update the Data
    - T3 applies for an Exclusive (X) lock to update copies_available → Blocked (Because T1 and T2 hold shared (S) locks)
  - Step 3: Transactions Release Locks
    - T1 and T2 finish reading and release their S locks
    - T3 gets the Exclusive (X) lock and updates copies_available to 2
    - T3 commits and releases the X lock.
  - Advantage: Allows multiple reads, ensuring better performance.
  - Disadvantage: Writers may be blocked if many readers hold shared locks.

Step-by-Step Execution Table (S and X Locks)

Step	T1 Operations	T2 Operations	T3 Operations	Lock Status on Book_ID 201
1	Requests **S-lock** to read Copies_Available ✅ **Granted**	-	-	S-Lock held by **T1**
2	Reading Copies_Available = 3	Requests **S-lock** to read ✅ **Granted**	-	S-Lock held by **T1 & T2**
3	-	Reading Copies_Available = 3	Requests **X-lock** to update ❌ **Blocked!**	S-Lock still held by **T1 & T2**
4	Finishes reading, releases **S-lock**	-	-	S-Lock still held by **T2**
5	-	Finishes reading, releases **S-lock**	-	🔓 **All locks released!**
6	-	-	Now gets **X-lock** ✅ **Granted**	X-Lock held by **T3**
7	-	-	Updates Copies_Available → 2	X-Lock still held by **T3**
8	-	-	Commits and releases **X-lock**	🔓 **No locks**

Summary

- Introduction to Concurrency Control
  - Ensures multiple transactions execute safely without conflicts or data corruption.
  - Preserves ACID properties (Atomicity, Consistency, Isolation, Durability).
  - Common concurrency issues:
    - Lost Updates: Two transactions update the same data, leading to inconsistencies.
    - Dirty Reads: A transaction reads uncommitted changes of another transaction.
    - Non-Repeatable Reads: A transaction reads different values at different times due to modifications by other transactions.
    - Phantom Reads: A transaction retrieves new rows inserted by another transaction when re-executing a query.
- Concurrency Control Techniques
  - Lock-Based Concurrency Control:
    - Uses Shared (S) and Exclusive (X) locks to control access.
    - Two-Phase Locking (2PL) ensures serializability but may lead to deadlocks.
  - Timestamp-Based Concurrency Control:
    - Assigns timestamps to transactions and ensures execution follows timestamp order.
    - Prevents conflicts but can cause frequent transactions restarts.
  - Optimistic Concurrency Control (OCC):
    - Transactions execute freely but validate before committing.
    - Efficient when conflicts are rare but may lead to high rollbacks in high-contention environments.
- Locking Mechanisms in DBMS
  - Locks prevent data inconsistencies and concurrency-related conflicts.
  - Types of Locks:
    - Binary Locks (Locked / Unlocked) - Simple but prevents parallel reads.
    - Shared (S) & Exclusive (X) Locks:

        - S Lock: Multiple transactions can read but not write.
        - X Lock: Only one transaction can read and write at a time.
- Example (S & X Locks in a Library Database):
    * T1 & T2 can both read using S locks.
    * T3 (wanting to update) must wait until T1 & T2 release locks.
    * Advantage: Allows multiple reads, improving performance.
    * Disadvantage: Writers may be blocked if many readers hold S locks.

Lecture 21

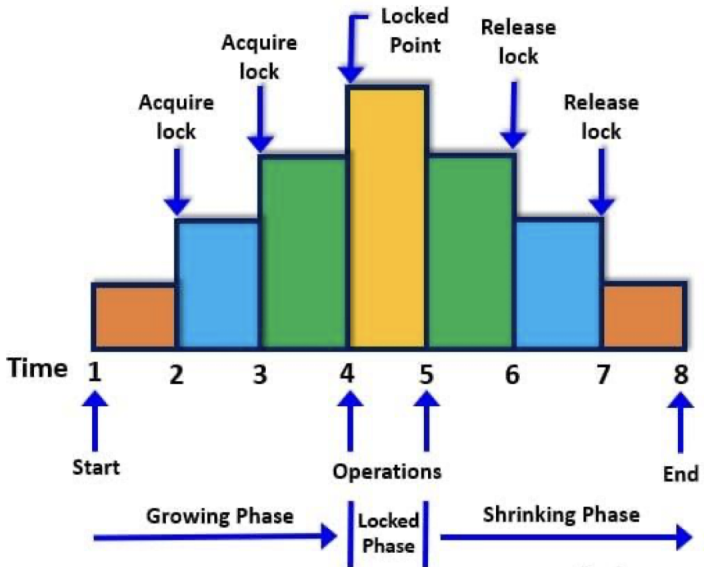
Outline

- Locking Protocols in Concurrency Control
- Timestamp-Based Concurrency Control
- Optimistic Concurrency Control (OCC)
- Comparison

Locking Protocols in Concurrency Control

- Locking protocols ensure data consistency and isolation in transactions.
- They prevent problems such as dirty reads, lost updates, and non-repeatable reads.
- Two-Phase Locking (2PL) Protocol: The Two-Phase Locking (2PL) Protocol is a widely used concurrency control technique that ensures serializability of transactions.
- A transaction in 2PL follows two distinct phases:

1. Growing Phase (Acquiring Locks)
    - A transaction can acquire locks (Shared or Exclusive) on data items.
    - It cannot release any locks in this phase.
2. Shrinking Phase (Releasing Locks)
    - A transaction releases locks that it acquired earlier.
    - It cannot acquire new locks in this phase.
    - The transaction eventually commits or rolls back.



- Variants of 2PL
  - Strict Two-Phase Locking (Strict 2PL)
    - Holds all exclusive (X) locks until the transaction commits or rolls back.
    - Prevents cascading aborts, where one transaction's rollback does not affect others.
    - Issue: Can cause longer waiting times, reducing concurrency.
    - Example: T2 had to wait until T1 committed, ensuring strict lock control.

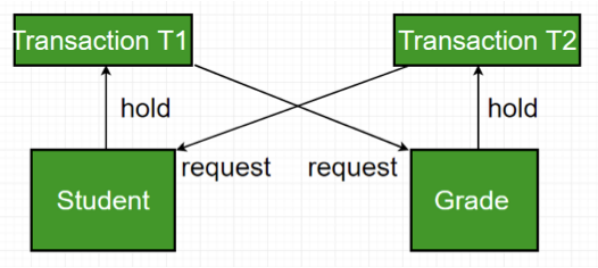
Step	T1 Operations	T2 Operations	Lock Status
1	Reads Data (S-Lock)	-	S-Lock acquired
2	Updates Data (X-Lock)	-	X-Lock acquired
3	-	Wants to read the same data (Blocked)	T1 holds X-Lock
4	T1 commits	-	Locks released
5	-	T2 can now proceed	-

  - Rigorous Two-Phase Locking (Rigorous 2PL)
    - Holds all locks (both Shared and Exclusive) until commit or rollback.
    - Stronger than Strict 2PL, as even Shared (S) Locks are held until the end.
    - Provides stronger isolation but reduces concurrency further.

Lock Type	Strict 2PL	Rigorous 2PL
Shared (S) Locks	Released normally	Held until commit/rollback
Exclusive (X) Locks	Held until commit/rollback	Held until commit/rollback
Prevents Cascading Aborts?	✅ Yes	✅ Yes
Concurrency Level	Higher than Rigorous	Lower than Strict

- Key Properties of 2PL:
  - Ensures serializability, meaning transactions execute in an order that produces a consistent final state.

– However, it can lead to deadlocks, where two or more transactions wait indefinitely for each other's locks to be released.



- Deadlocks in 2PL:
  – Deadlocks occur when two or more transactions wait indefinitely for each other's locks to be released.

Step	T1 Operations	T2 Operations	Lock Status
1	Acquires X-Lock on A	-	T1 locks A
2	-	Acquires X-Lock on B	T2 locks B
3	Requests X-Lock on B (Blocked)	-	T1 waiting for B
4	-	Requests X-Lock on A (Blocked)	T2 waiting for A
5	**Deadlock!**	**Deadlock!**	Both transactions are waiting for each other

  – Solution to Deadlocks:
    * Deadlock Detection → Identify deadlocks using wait-for graphs and abort one transaction.
    * Deadlock Prevention → Use timeouts or assign priorities to transactions to avoid cyclic waiting.

- Summary of Locking Protocols

Locking Protocol	Lock Behavior	Prevents Cascading Aborts?	Concurrency Impact	Deadlock Risk?
Basic 2PL	Acquires locks, then releases them	❌ No	High	✅ Yes
Strict 2PL	Holds X-locks until commit/rollback	✅ Yes	Medium	✅ Yes
Rigorous 2PL	Holds all locks until commit/rollback	✅ Yes	Low	✅ Yes

Timestamp-Based Concurrency Control

- What is Timestamp-Based Concurrency Control?
  – Timestamp-Based Concurrency Control (TBCC) is a technique used in database management systems to ensure serializability of transactions without using locks.

- Instead of locking data items, TBCC assigns each transaction a unique timestamp and ensures transactions execute in a timestamp order.

- Key Idea:
  - Each transaction is assigned a timestamp (TS) when it begins.
  - The system ensures that transactions execute in timestamp order to prevent conflicts.
  - Older transactions execute before newer ones to maintain consistency.

- Key Components in TBCC
  - Every data item (X) in the database maintains two timestamps:
    - Read Timestamp (RTS(X)) → the latest timestamp of any transaction that has read X.
    - Write Timestamp (WTS(X)) → the latest timestamp of any transaction that has write X.
  - These timestamps help the system decide whether a new read or write request should be allowed or rejected.

- Basic Timestamp Ordering Protocol
  - The protocol follows two main rules for handling read and write requests.
  - Read Operation (T wants to read X)
    - If TS(T) <WTS(X) → Abort T (This prevents reading outdated data).
    - Else, allow T to read X and update RTS(X) = max(RTS(X), TS(T))
  - Write Operation (T wants to write X)
    - If TS(T) <RTS(X) → Abort T (This prevents overwriting newer data).
    - If TS(T) <WTS(X) → Abort T (T is outdated and trying to overwrite a newer value).
    - Else, allow T to write X and update WTS(X) = TS(T).

- Example of Timestamp-Based Concurrency Control
  - Consider two transactions (T1 and T2) with timestamps:
    - T1 (TS = 5)
    - T2 (TS = 10)

Step	T1 (TS=5) Operations	T2 (TS=10) Operations	RTS(X)	WTS(X)	Action Taken
1	Read(X)	-	5	0	Allowed (RTS updated to 5)
2	-	Write(X)	5	0	Allowed (WTS updated to 10)
3	Write(X)	-	5	10	**T1 Aborted (TS=5 < WTS=10)**

  - Explanation:
    - T1 reads X first, so RTS(X) = 5.
    - T2 writes X next, so WTS(X) = 10.
    - When T1 later tries to write X, it is aborted because its timestamp (TS = 5) is older than WTS(X) = 10.
    - Ensures serializability by preventing outdated transactions from modifying newer values.

- Thomas Write Rule (Optimization)
  - The Thomas Write Rule is an optimization used in timestamp ordering.
  - Instead of aborting a transaction when TS(T) <WTS(X), the system ignores the write (instead of rolling back).

- Thomas Write Rule Handling for Writes:
  - If TS(T) <RTS(X) → Abort T (Prevents stale writes).
  - If TS(T) <WTS(X) → Ignore the write (Instead of aborting).
  - Otherwise, Allow the write and update WTS(X) = TS(T).
- Example:
  - T1 (TS = 5) tries to write X when WTS(X) = 10.
  - Instead of aborting, we ignore the write because a newer value (WTS = 10) already exists.
  - This improves concurrency while still maintaining consistency.
- Advantages and Disadvantages of TBCC

Aspect	Advantages	Disadvantages
Concurrency	Higher concurrency than locking protocols.	Can cause **frequent aborts** in high-contention cases.
Deadlocks	**No deadlocks** since transactions do not wait.	-
Performance	Works well in **read-heavy workloads.**	**High abort rates** in write-intensive applications.
Cascading Aborts	**No cascading aborts** since transactions are aborted immediately.	-

- No deadlocks occur, but frequent aborts can happen in high-contention environments.
- Best suited for read-heavy databases where transactions rarely conflict.

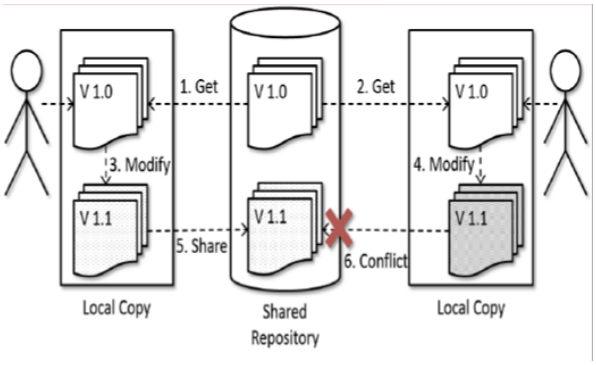
Optimistic Concurrency Control (OCC)

- What is Optimistic Concurrency Control (OCC)?
  - Optimistic Concurrency Control (OCC) is a concurrency control method used in database management systems where transaction execute without acquiring locks and are only checked for conflicts at the end of execution.
- Key Idea:
  - Transactions execute optimistically, assuming that conflicts are rare.
  - Instead of locking resources, OCC validates transactions before committing.
  - If a conflict is detected, the transaction is aborted and restarted.
- Best suited for systems with low contention (few conflicts), such as read-heavy applications.
- Not ideal for high-contention environments where frequent conflicts cause high abort rates.
- OCC is divided into three main phases:

1. Read Phase (Execution Phase)
   - The transaction reads data from the database into a local copy without acquiring locks.
   - All changes are stored in a temporary workspace (buffer) instead of the database.
   - The database remains unchanged during this phase.
2. Validation Phase (Conflict Detection)
   - Before committing, the system checks whether conflicts occurred with other concurrent transactions.

- Each transaction is validated to ensure serializability.
- If a conflict is detected, the transaction is aborted and restarted.

3. Write Phase (Commit Phase)
   - If validation succeeds, changes from the transaction's local copy are applied to the database.
   - If validation fails, the transaction is rolled back and restarted.



- OCC Validation Rules: Validation in OCC ensures that transactions execute serializability.
- Each transaction Ti is assigned:
  - Start Timestamp (TS_start(Ti)) → When Ti starts executing.
  - Validation Timestamp (TS_validate(Ti)) → When Ti enters the validation phase.
  - Commit Timestamp (TS_commit(Ti)) → When Ti successfully commits.
- Validation Conditions (for a transaction Tj begin validated against an already committed transaction Ti):

1. If Ti completes before before Tj starts
   - No conflict (Tj can commit)
   - Condition TS_commit(Ti) <TS_start(Tj)
2. If Tj starts before Ti commits and Tj reads data modified by Ti
   - Conflict (Tj is aborted)
   - Condition TS_start(Tj) <TS_commit(Ti) <TS_validate(Tj)
3. If Tj writes to the same data item asa Ti before Ti commits
   - Conflict (One transaction must abort)

- Example: Consider two transactions, T1 and T2, that both access a bank account
- Scenario: A bank account balance is stored in a table
- T1 reads the balance and updates it
- T2 reads the same balance and updates it before T1 commits

Step	T1 (TS = 5) Operations	T2 (TS = 10) Operations	Validation Check	Action
1	Reads Balance = 500	-	-	Allowed
2	-	Reads Balance = 500	-	Allowed
3	Updates Balance to 600 (local copy)	-	-	-
4	-	Updates Balance to 700 (local copy)	-	-
5	**Validation Check for T1**	-	✅ Passes (No conflicts)	T1 commits
6	-	**Validation Check for T2**	❌ Fails (T1 modified the same data)	T2 Aborted

- Ensures serializability without using locks
- Advantages and Disadvantages of OCC

Feature	Advantages	Disadvantages
Concurrency	High concurrency since no locks are used.	Frequent aborts in high-contention workloads.
Deadlocks	No deadlocks (since there are no locks).	-
Performance	Works well in read-heavy workloads.	Poor performance in write-heavy workloads.
Resource Usage	No locking overhead, reducing memory use.	More CPU usage due to frequent transaction restarts.

- OCC is best suited for:
  - Read-heavy applications (e.g., analytics, reporting systems).
  - Low-contention environments (few simultaneous transactions modifying the same data)
  - Distributed databases where locking is expensive.
- Avoid OCC if:
  - Many transactions update the same data frequently (high contention)
  - Frequent aborts negatively impact performance.

Comparison

Feature	Lock-Based Concurrency Control	Timestamp-Based Concurrency Control	Optimistic Concurrency Control (OCC)
Approach	Uses **locks** (Shared & Exclusive) to control access.	Uses **timestamps** to order transactions.	Assumes conflicts are rare and validates transactions at commit time.
Phases	Transactions acquire locks before accessing data.	Transactions are ordered based on timestamps and checked before execution.	Transactions go through **Read, Validation, and Write** phases.
Conflict Resolution	Conflicts cause blocking (waiting) or deadlocks.	Conflicts are resolved by **aborting** and restarting transactions.	Transactions are validated at commit time; if conflicts occur, the transaction is **aborted and restarted**.
Deadlocks	**Possible** (due to circular waiting on locks).	**Not possible** (strict ordering using timestamps).	**Not possible** (no locks are used).
Read Consistency	Ensured using locks (but can lead to waiting).	Ensured through timestamp ordering.	Ensured after validation phase.
Write Conflicts	If another transaction holds an **Exclusive (X) Lock**, the second transaction **waits**.	If a newer transaction modifies a value **before** an older one commits, the older one **aborts**.	If conflicts are found in the validation phase, the transaction **aborts** and restarts.

Feature	Lock-Based Concurrency Control	Timestamp-Based Concurrency Control	Optimistic Concurrency Control (OCC)
Best for Read-Heavy Workloads?	❌ **No** (Locks can block reads).	✅ **Yes** (Reads are quick, no locks).	✅ **Yes** (Reads happen without locks).
Best for Write-Heavy Workloads?	✅ **Yes** (Efficient if managed properly).	❌ **No** (Frequent aborts on conflicts).	❌ **No** (High abort rates when contention is high).
Overhead	High (due to locking and deadlock handling).	Medium (timestamps must be managed, but no locks).	Low (for low contention) but high for high-contention workloads (due to frequent aborts).
Transaction Abort Rate	**Low** (Blocked transactions wait rather than abort).	**Medium** (Aborts on conflict, but predictable).	**High** (Many transactions may be restarted in high-contention environments).
Performance in High Contention	❌ **Can slow down due to locking overhead.**	✅ **Predictable but may cause many aborts.**	❌ **Poor (frequent restarts cause performance degradation).**
Performance in Low Contention	✅ **Efficient (Few locks, minimal blocking).**	✅ **Efficient (Timestamps avoid unnecessary waiting).**	✅ **Best choice (Few conflicts = fewer aborts).**
Use Cases	**General-purpose DBMS** (OLTP, banking, airline reservations).	**Databases with strict ordering requirements** (Distributed systems, logging, history tracking).	**Read-heavy workloads** (Data warehouses, reporting systems, analytics).

Summary

- Locking Protocols in Concurrency Control
  - Ensure data consistency and isolation by preventing issues like dirty reads and lost updates.
  - Two-Phase Locking (2PL) is a common technique that ensures serializability:
    - Growing Phase: Acquires locks but cannot release them.
    - Shrinking Phase: Releases locks but cannot acquire new ones.
  - Variants of 2PL:
    - Strict 2PL: Holds all exclusive locks until commit / rollback (prevents cascading aborts).
    - Rigorous 2PL: Holds all locks (shared & exclusive) until commit / rollback (strongest isolation).

  - Deadlocks in 2PL:
    * Occur when two transactions wait indefinitely for each other's locks.
    * Solutions: Deadlock detection (abort one transaction) or prevention (timeouts, priorities).

- Timestamp-Based Concurrency Control (TBCC)
  - Uses timestamps instead of locks to maintain serializability.
  - Each data item maintains:
    * Read Timestamp (RTS) → Latest timestamp of a transaction that read the item.
    * Write Timestamp (WTS) → Latest timestamp of a transaction that wrote the item.
  - Rules: Transactions are aborted if they read outdated data or overwrite newer values.
  - Thomas Write Rule (Optimization):
  - Advantages: No deadlocks, efficient for read-heavy databases.
  - Disadvantages: Frequent transaction aborts in high-contention environments.

- Optimistic Concurrency Control (OCC)
  - Assumes conflicts are rare and validates transactions before committing instead of using locks.
  - Phases:
    * Read Phase: Transaction reads data into a local copy.
    * Validation Phase: System checks for conflicts with other transactions.
    * Write Phase: If validation succeeds, changes are applied to the database.
  - Validation Conditions ensure serializability.
  - Best suited for: Read-heavy applications (e.g., analytics, distributed databases).
  - Not ideal for: High-contention systems, where frequent conflicts cause high abort rates.

- Comparison of Concurrency Control Techniques

Method	**Advantages**	**Disadvantages**
2PL	Ensures serializability	Can cause deadlocks
TBCC	No deadlocks, good for reads	Frequent transaction aborts
OCC	High performance in low contention	High abort rates in conflicts

Lecture 22

Outline

- Introduction to NoSQL Databases
- Types of NoSQL Databases
- NoSQL vs. Relational Databases
- Practical Applications

Introduction to NoSQL Databases

- What is NoSQL?
  - "Not Only SQL" - A class of databases designed to handle large-scale, unstructured, or semi-structured data.
  - Developed to overcome limitations of traditional relational databases in terms of scalability, flexibility, and performance.
- Characteristics of NoSQL Databases
  - Schema-less or flexible schema: Unlike relational databases, NoSQL does not require predefined schemas.
  - Horizontal Scalability: Designed for distributed systems using multiple servers.
  - High Availability: Replication and partitioning techniques ensure fault tolerance.
  - Optimized for Specific Workloads: Tailored for high-speed transactions, large-scale analytics, or complex relationships.
- When to Use NoSQL?
  - Big Data Applications (e.g., social media, IoT, real-time analytics).
  - Applications requiring scalability and fast reads/writes.
  - Situations where flexibility in schema is necessary.

Types of NoSQL Databases

1. Key-Value Databases
   - Concept:
     - Data is stored as a collection of key-value pairs (similar to a dictionary or hashmap).
     - Optimized for fast lookup, caching, and session management.
   - Examples:

  - Redis - Used for caching, real-time analytics, session storage.
  - Amazon DynamoDB - Scalable NoSQL database for web applications.
- Pros & Cons
  - Fast retrieval with low latency.
  - Scales easily for large amounts of simple data.
  - Not suitable for complex queries (e.g., joins, relationships).

2. Document Databases

- Concept:
  - Stores semi-structured data in JSON, BSON, or XML format.
  - Each document is a self-contained unit, including its own schema.
- Examples:
  - MongoDB - Popular for web applications, flexible schema.
  - CouchDB - Uses JSON documents with RESTful API.
- Pros & Cons
  - Flexible schema (good for applications involving data structures).
  - Rich querying support (can retrieve nested data).
  - Not optimized for complex transactions (like ACID compliance in relational DBs).

3. Columnar Databases

- Concept:
  - Stores data in columns instead of rows, improving read efficiency for analytical queries.
  - Used in data warehouses and real-time analytics.
- Examples:
  - Apache Cassandra - Distributed, fault-tolerant, optimized for fast writes.
  - Google Bigtable - Scalable column store used in Google Analytics.
- Pros & Cons
  - High write and read performance for large datasets.
  - Efficient for aggregation queries (e.g., SUM, AVG).
  - Not ideal for transactional applications requiring ACID compliance.

4. Graph Databases

- Concept:
  - Designed for storing relationships efficiently as a graph structure (nodes and edges).
  - Best suited for social networks, recommendation engines, fraud detection.
- Examples:
  - Neo4j - Leading graph database with Cypher query language.
  - Amazon Neptune - Fully managed graph database service.
- Pros & Cons
  - Best for handling complex relationships.
  - Highly efficient traversal queries.
  - Not optimized for large-scale structured data processing.

- Key Differences

Feature	Relational Databases (SQL)	NoSQL Databases
Data Model	Tables (Structured)	Flexible models (Key-Value, Document, Column, Graph)
Schema	Fixed (Predefined Schema)	Dynamic (Schema-less)
Scalability	Vertical (Scaling Up)	Horizontal (Scaling Out)
Query Language	SQL (Structured Query Language)	Varies (No unified query language)
Transaction Support	ACID (Strong Consistency)	BASE (Eventual Consistency)
Best Use Cases	Structured business data	Big Data, Real-Time Processing, Flexible Schema Applications

- When to use SQL vs. NoSQL?
  - SQL Databases are best for:
    - Structured Data (e.g., financial systems, ERP, CRM).
    - Applications requiring strong consistency and ACID transactions.
    - Complex queries with multiple table joins.
  - NoSQL Databases are best for:
    - Big Data applications (social media, IoT, streaming data).
    - Applications needing high scalability and availability.
    - Flexible schema requirements (e.g., e-commerce product catalogs).
- Real-World Use Cases of NoSQL Databases
  - E-commerce (Amazon DynamoDB, MongoDB) - Handling product catalogs and real-time pricing.
  - Social Media (Cassandra, Neo4j) - Storing user connections and relationships.
  - Real-Time Analytics (Bigtable, Apache HBase) - Processing large-scale data streams.
  - Recommendation Engines (Neo4j, Redis) - Graph-based data storage for AI-powered recommendations.
- Sample command in MongoDB:
  - db.students.insert({name:“Alice”, age:22, major:“Computer Science”})
  - db.students.find({major:“Computer Science”})

Summary

- Introduction to NoSQL
  - “Not Only SQL” - Designed for scalability, flexibility, and performance over traditional relational databases.
  - Key Features:
    - Schema-less or flexible schema.
    - Horizontally scalable for distributed systems.
    - High availability through replication.
    - Optimized for specific workloads (e.g., fast transactions, large-scale analytics).
  - Best suited for:
    - Big Data (social media, IoT, real-time analytics).

    * Applications needing scalability and fast reads / writes.
    * Situations requiring flexible schemas.

- Types of NoSQL Databases
  - Key-Value Databases
    * Structure: Stores data as key-value pairs (like a dictionary).
    * Examples: Redis (caching, session storage), DynamoDB (web applications).
    * Pros: Fast retrieval, scalable.
    * Cons: Not suited for complex queries.
  - Document Databases
    * Structure: Stores semi-structured data (JSON, BSON, XML).
    * Examples: MongoDB (web apps), CouchDB (RESTful API).
    * Pros: Flexible schema, supports nested queries.
    * Cons: Not ACID compliant.
  - Columnar Databases
    * Structure: Stores data in columns instead of rows (ideal for analytics).
    * Examples: Apache Cassandra (fast writes), Google Bigtable (large-scale analytics).
    * Pros: High read / write performance for large datasets.
    * Cons: Not suitable for ACID transactions.
  - Graph Databases
    * Structure: Uses nodes and edges to store relationships efficiently.
    * Examples: Neo4j (social networks, recommendation engines), Amazon Neptune.
    * Pros: Best for complex relationships.
    * Cons: Not ideal for structured data processing.

- NoSQL vs. Relational Databases

Feature	**SQL Databases**	**NoSQL Databases**
Structure	Fixed schema	Flexible schema
Scalability	Vertical (single server)	Horizontal (multiple servers)
Transactions	Strong consistency (ACID)	Eventual consistency (BASE)
Best for	Financial systems, ERP, CRM	Big Data, real-time applications

- Practical Applications
  - E-commerce (MongoDB, DynamoDB) - Product catalogs, real-time pricing.
  - Social Media (Cassandra, Neo4j) - Storing user relationships.
  - Real-Time Analytics (Bigtable, Apache HBase) - Processing large data streams.
  - Recommendation Engines (Neo4j, Redis) - AI-powered recommendations.

Lecture 23

Outline

- Introduction to Distributed Databases and Big Data
- What is a Distributed Database?
- CAP Theorem
- Partitioning and Replication
- Impact of Database Design on Big Data Analytics

Introduction to Distributed Databases and Big Data

- Definition of Big Data:
  - Extremely large datasets that require advanced storage, processing, and analysis techniques.
  - Characterized by the 3 V's: Volume, Velocity, and Variety (sometimes expanded to include Veracity and Value).
- Traditional Databases vs. Big Data:
  - Traditional Databases: Relational databases, structured data, SQL queries, centralized storage.
  - Big Data Systems: Distributed storage, NoSQL databases, schema flexibility, real-time analytics.
- Challenges in Handling Big Data:
  - Scalability, storage, processing speed, fault tolerance, consistency, and security.

What is a Distributed Database?

- Definition: A database that is distributed across multiple locations or computing nodes.
- Why Use Distributed Databases?
  - Improved performance
  - High availability
  - Scalability
  - Fault tolerance
- Types of Distributed Databases:
  - Homogeneous Distributed Databases: All nodes use the same DBMS.
  - Heterogenous Distributed Databases: Different DBMSs across nodes.
  - Federated Databases: Multiple independent databases functioning as one system.

CAP Theorem

- Proposed by Eric Brewer, the CAP Theorem states that in a distributed system, only two out of the three properties can be guaranteed at a time:

1. Consistency (C): Every read receives the most recent write or an error.
2. Availability (A): Every request receives a response, whether it is the latest update or not.
3. Partition Tolerance (P): The system continues to operate despite network failures.

- Implications of CAP Theorem:
  - A system must trade-offs depending on its use case.
  - CA (Consistency & Availability, but not Partition Tolerant): Traditional relational databases.
  - CP (Consistency & Partition Tolerant, but not Available): Google's BigTable, HBase.
  - AP (Availability & Partition Tolerant, but not Consistent): DynamoDB, Cassandra.

Partioning and Replication

- Partitioning in Distributed Databases
  - Definition: Dividing a large database into smaller, manageable parts across different nodes.
  - Types of Partitioning:
    - Horizontal Partitioning (Sharding): Each partition stores a subset of rows.
    - Vertical Partitioning: Each partition contains specific columns of data.
    - Key-Based Partitioning: Using a hash function to distribute data across nodes.
    - Range-Based Partitioning: Dividing data into partitions based on ranges (e.g., dates, numeric values).
  - Advantages of Partitioning:
    - Improves performance by reducing query load on individual nodes.
    - Enhances scalability and parallel processing.
    - Reduces network congestion and improves fault tolerance.
- Replication in Distributed Databases
  - Definition: Keeping multiple copies of the same data on different nodes to improve availability and fault tolerance.
  - Types of Replication:
    - Master-Slave Replication: One primary database (master) updates read-only copies (slaves).
    - Multi-Master Replication: Multiple masters synchronize data updates.
    - Peer-to-Peer Replication: All nodes are equal, allowing flexible read and write operations.
  - Trade-offs in Replication:
    - More copies increase redundancy and availability but may impact consistency.
    - Synchronization overhead affects performance.

Impact of Database Design on Big Data Analytics

- Choosing Between SQL and NoSQL for Big Data:
  - SQL (Relational Databases): Ensure ACID properties but lacks scalability.
  - NoSQL (Key-Value, Columnar, Document, Document, Graph): Prioritizes scalability, flexibility, and performance.
- How Partitioning and Replication AFfect Analytics:
  - Partitioning improves query performances and parallel processing.
  - Replication ensures high availability and fault tolerance, critical for real-time analytics.
- Trade-offs in Performance and Consistency:
  - Strong Consistency: Ensures data correctness but increases latency.
  - Eventual Consistency: Improves speed but may result in temporary stale reads.
- Case Studies:
  - Google BigTable (CP System)
  - Amazon DynamoDB (AP System)
  - Facebook's TAO for Social Graphs (Optimized Graph Database)

Summary

- Introduction to Distributed Databases & Big Data
  - Big Data: Extremely large datasets requiring advanced storage and processing. Defined by Volume, Velocity, Variety (+ Veracity & Value).
  - Traditional Databases vs. Big Data:
    - Traditional: Structured, centralized, SQL-based.
    - Big Data: Distributed, NoSQL, real-time analytics.
  - Challenges: Scalability, storage, processing speed, fault tolerance, consistency, security.
- What is a Distributed Database?
  - A database spread across multiple nodes for better performance, availability, scalability, and fault tolerance.
  - Types:
    - Homogeneous: All nodes use the same DBMS.
    - Heterogeneous: Different DBMS across nodes.
    - Federated: Independent databases functioning as one system.
- CAP Theorem (Consistency, Availability, Partition Tolerance)
  - In distributed systems, only two of the three can be achieved at a time:
    - C (Consistency): Every read gets the latest write or an error.
    - A (Availability): Every request gets a response, even if outdated.
    - P (Partition Tolerance): The system remains operational despite network failures.
  - Examples:
    - CA: Traditional relational databases.
    - CP: Google BigTable, HBase.

        * AP: DynamoDB, Cassandra.

- Partitioning & Replication
  - Partitioning (Dividing data across nodes):
    * Horizontal (Sharding): Splitting rows.
    * Vertical: Splitting columns.
    * Key-Based: Using a hash function for distribution.
    * Range-BAsed: Partitioning based on value ranges (e.g., dates).
  - Benefits: Enhances performance, scalability, and fault tolerance.
  - Replication (Storing multiple copies across nodes):
    * Master-Salve: One primary database updates read-only copies.
    * Multi-Master: Multiple nodes synchronize updates.
    * Peer-to-Peer: All nodes are equal.
  - Trade-offs: Higher availability but potential consistency issues and performance overhead.
- Impact on Big Data Analytics
  - SQL vs. NoSQL: SQL ensures ACID compliance but lacks scalability; NoSQL offers high scalability and performance.
  - Partitioning & Replication Benefits:
    * Partitioning boosts query speed and enables parallel processing.
    * Replication ensures high availability for real-time analytics.
  - Performance vs. Consistency Trade-offs:
    * Strong consistency = Correct data but higher latency.
    * Eventual consistency = Faster responses with temporary stale reads.
- Case Studies
  - Google BigTable (CP System)
  - Amazon DynamoDB (AP System)
  - Facebook TAO (Graph database optimized for social networks)

Lecture 24

Outline

- Introduction to Database Security
- Access Control Mechanisms
- Authentication Methods
- Encryption Techniques in Databases
- Best Practices for Securing Databases

Importance of Database Security in Modern Systems

- Critical Role of Databases: Nearly all modern applications, from banking and healthcare to social media and e-commerce, rely on databases to store sensitive data.
- Growing Security Threats: With increasing cyber-attacks, protecting databases is a top priority for organizations.
- Regulatory Compliance: Laws such as GDPR (General Data Protection Regulation), HPIAA (Health Insurance Portability and Accountability Act) and PCI-DSS (Payment Card Industry Data Security Standard) enforce strict security requirements for databases.

Real-World Consequences of Security Breaches

- Security breaches can lead to:
  - Financial Loss: Organizations face direct costs (e.g., fines, compensations) and indirect costs (e.g., loss of customers, lawsuits).
  - Reputation Damage: Customers lose trust, leading to declining business.
  - Legal and Regulatory Issues: Non=compliance can lead to heavy fines and sanctions.
- Case Studies:
  - Equifax Data Breach (2017): Exposed 147 million users' sensitive data due to an unpatched vulnerability in a web application. Cost: over $1.4 billion.
  - Yahoo Data Breach (2013-2014): Compromised 3 billion accounts due to poor security measures, leading to a massive trust deficit and financial losses.
  - Facebook Data Leak (2021): Personal information of 533 million users leaked due to a misconfigured database.

Overview of Security Principles: The CIA Triad

- Database security is built upon three fundamental principles:
- Confidentiality (Who can access the data?)
  - Ensuring that only authorized users have access to data.
  - Example: Encryption and Access Control Mechanisms.
- Integrity (How do we ensure the data is accurate and trustworthy?)
  - Protecting data from unauthorized modification or deletion.
  - Example: Checksums, Digital Signatures, and Database Constraints.
- Availability (How do we ensure the data is accessible when needed?)
  - Ensuring data remains accessible during cyber-attacks or system failures.
  - Example: Backup Strategies, Load Balancing, and Failover Mechanisms.

Access Control Mechanisms

- Purpose: Ensuring only authorized users have access to database resources.
- Types of Access Control:
  - Discretionary Access Control (DAC) - Example: User permissions in MySQL & PostgreSQL.
  - Mandatory Access Control (MAC) - Used in highly secure environments, e.g., military databases.
  - Role-Based Access Control (RBAC) - Example: Implementing roles in Oracle and SQL Server.
- Real-World Example:
  - The 2019 Capital One breach: Poor access control led to exposure of millions of records.
  - Solution: Implementing least privilege access and better auditing mechanisms.

Authentication Methods

- Importance: Ensuring only legitimate users access the database.
- Types of Authentication:
  - Password-based authentication (Example: MySQL, PostgreSQL user authentication).
  - Multi-Factor Authentication (MFA) (Example: Google Cloud SQL requiring MFA).
  - Certificate-based authentication (Example: SSL/TLS authentication in MongoDB).
- Real-World Example:
  - Yahoo's 2013 data breach: Weak authentication led to the exposure of 3 billion accounts.
  - Solution: Stronger password policies, enforcing MFA, and password hashing.

Encryption Techniques in Databases

- Importance: Protecting data at rest in transit.
- Types of Encryption:
  - Data at Rest Encryption: Transparent Data Encryption (TDE) in SQL Server, Oracle, and MySQL.
  - Data in Transit Encryption: Use of SSL/TLS for encrypting database connections.
  - Column-level Encryption: Example: Encrypting sensitive fields like SSNs or credit card numbers.
- Real-World Example:
  - Equifax Data Breach (2017): Unencrypted sensitive information led to exposure of 147 million records.
  - Solution: Encrypting all sensitive data and enforcing secure transmission protocols.

Best Practices for Securing Databases

1. Implementing the principle of least privilege (POLP).
2. Regular database updates and patching (Example: Log4j vulnerability in 2021).
3. Auditing and logging database activities.
4. Using firewalls and Intrusion Detection Systems (IDS).
5. Regular backups and disaster recovery planning.

Case Study: LinkedIn's 2012 Data Breach

- Overview of the Incident
  - In 2012, LinkedIn, a major professional networking platform, suffered a data breach where approximately 6.5 million user passwords were stolen and leaked online.
  - The breach was initially thought to be limited in scope, but in 2016, further investigation revealed that 167 million LinkedIn accounts had been compromised.
- Root Cause: Improper Password Encryption
  - Use of Weak Hashing Algorithm:
    - LinkedIn stored passwords using SHA-1 hashing without a cryptographic salt.
    - SHA-1 was already considered weak at the time, and without a salt, it made the hashed passwords vulnerable to rainbow table attacks.
  - How Attackers Exploited It:
    - Hackers gained access to LinkedIn's user database and extracted password hashes.
    - Using brute-force attacks and rainbow tables, they were able to crack a significant number of these passwords.
- Consequences of the Breach
  - Massive Data Exposure:
    - In 2016, a hacker known as "Peace" put 167 million LinkedIn accounts, including email addresses and passwords, up for sale on the dark web.
  - Loss of User Trust:

        * The breach exposed LinkedIn's weak security practices, damaging its reputation.
    - Legal and Financial Repercussions:
        * LinkedIn faced class-action lawsuits and was forced to improve its security measures.

- Lessons Learned and Security Improvements
    - Stronger Password Hashing:
        * LinkedIn switched to bcrypt, a much stronger cryptographic hashing function that includes salting and multiple rounds of hashing to make brute-force attacks difficult.
    - Mandatory Password Resets:
        * LinkedIn forced all affected users to reset their passwords to prevent further account compromises.
    - Multi-FActor Authentication (MFA):
        * Users were encouraged to enable two-factor authentication (2FA) to add an extra layer of security.
    - Improved Security Monitoring:
        * The company implemented real-time monitoring and intrusion detection to identify suspicious activity earlier.

Summary

- Importance of Database Security
    - Databases store critical data for banking, healthcare, social media, and e-commerce.
    - Rising cyber threats make database security essential.
    - Regulations (e.g., GDPR, HIPAA, PCI-DSS) enforce strict security measures.
- Consequences of Security Breaches
    - Financial loss (fines, lawsuits, customer loss).
    - Reputation damage (loss of trust).
    - Legal & compliance issues (penalties for non-compliance).
    - Case Studies:
        * Equifax (2017) - Exposed 147M users due to an unpatched vulnerability.
        * Yahoo (2014-2014) - 3B accounts compromised due to weak security.
        * Facebook (2021) - 533M user records leaked due to misconfiguration.
- Security Principles: The CIA Triad
    - Confidentiality - Prevent unauthorized access (e.g., encryption, access control).
    - Integrity - Ensure data accuracy (e.g., checksums, digital signatures).
    - Availability - Ensue data is accessible (e.g., backups, load balancing).
- Access Control Mechanisms
    - Discretionary Access Control (DAC) - User-defined permissions.
    - Mandatory Access Control (MAC) - Strict access rules for high-security systems.
    - Role-Based Access Control (RBAC) - Assigns permissions based on roles.
    - Case Study: Capital One (2019) - Poor access control exposed millions of records.
- Authentication Methods

  - Password-based authentication - Standard but vulnerable.
  - Multi-Factor Authentication (MFA) - Adds security layers (e.g., Google Cloud SQL).
  - Certificate-based authentication - Uses SSL/TLS for secure access.
  - Case Study: Yahoo (2013) - Weak authentication exposed 3B accounts.
- Encryption Techniques
  - Data at Rest Encryption - Protects stored data (e.g., TDE in SQL Server).
  - Data in Transit Encryption - Secure communication (e.g., SSL / TLS).
  - Column-Level Encryption - Encrypts sensitive fields like SSNs.
  - Case Study: Equifax (2017) - Lack of encryption led to 147M records being exposed.
- Best Practices for Securing Databases
  - Principle of Least Privilege (POLP) - Limit user access.
  - Regular updates & patching - Prevent exploits (e.g., Log4j vulnerability).
  - Auditing & logging - Track database activities.
  - Firewalls & Intrusion Detection Systems (IDS) - Block unauthorized access.
  - Backups & disaster recovery - Ensure data availability.
- Case Study: LinkedIn (2012) Data Breach
  - Issue: Stored passwords with weak SHA-1 hashing (no salt).
  - Hackers used brute-force attacks to crack 167M passwords.
  - Consequences: Data sold on the dark web, loss of user trust, lawsuits.
  - Lessons Learned:
    * Stronger hashing (bcrypt).
    * Mandatory password resets.
    * Encouraging MFA.
    * Improved security monitoring.