DMC 8103

MASTER OF COMPUTER APPLICATION

DATABASE TECHNOLOGY

CENTRE FOR DISTANCE EDUCATION

ANNA UNIVERSITY

CHENNAI – 600 025

Course Writer

Dr. Radha Senthilkumar

Associate Professor

Department of Information Technology

MIT Campus

Anna University

Chennai – 600044.

Course Reviewer

Dr. Geetha P

Associate Professor

Department of Information Science and Technology

CEG Campus

Anna University

Chennai – 600025.

DATABASE TECHNOLOGY

COURSE OBJECTIVES

Upon Completion of the course, the students should be able to:

- Understand the importance of Modelling an Entity Relationship Diagram, Map the Entity RelationshipDiagram to Relations and Database Normalization.
- Gain Knowledge on Designing Parallel Databases and Distributed Databases.
- Understand the Basics of XML Databases, Web Databases, Active Databases and TemporalDatabases.
- Gain Basic Knowledge on MongoDB NoSQL Database.
- Understand the Basics of Data Warehousing and Data Mining.

UNIT I **RELATIONAL MODEL:** Entity Relationship Model – Relational Data Model – Mapping Entity Relationship Model to Relational Model – Relational Algebra – Structured Query Language –Database Normalization – First Normal Form – Second Normal Form – Third Normal Form – Boyce Codd Normal Form – Fourth Normal Form – Fifth Normal Form.

UNIT II **PARALLEL AND DISTRIBUTED DATABASES**: Parallel Databases – I/O Parallelism – Inter-Query and Intra-Query Parallelism – Inter-Operation and Intra-Operation Parallelism –Distributed Database Architecture – Distributed Data Storage – Distributed Transactions – Distributed Query Processing – Distributed Transaction Management – ACID Properties – Concurrency Control.

UNIT III **XML DATABASES,WEB DATABASES,ACTIVEDATABASES AND TEMPORAL**

DATABASES: XML Data Model – DTD – XML Schema – XML Querying – Web Databases – Open Database Connectivity – Java Database Connectivity – Accessing Relational Database using PHP – Event Condition Action Model – Design and Implementation Issues for Active Databases – Temporal Databases – Interpreting Time in Relational Databases.

UNIT IV **NoSQL DATABASES**: NoSQL Database vs.SQL Databases– CAP Theorem –Migrating from RDBMS to NOSQL – MongoDB – CRUD Operations– MongoDB Sharding – MongoDB Replication – Web Application Development using MongoDB with PHP and Java.

UNIT V DATA WAREHOUSING AND DATA MINING: Data Warehouse – Characteristics – Three Tier Architecture – Data Cube – Online Analytical Processing vs. Online Transaction Processing – Online Analytical Processing Operations – Star Schema – Snow Flake Schema – Fact Constellation Schema – Data Mart – Data Mining – Apriori Algorithm for Association Rule Mining – Decision Tree Induction using Information Gain for Classification – k-Means Clustering.

COURSE OUTCOMES

On completion of the course, the student will be able to

- Design a Relational Database for an Enterprise.
- Design a Parallel Database and Distributed Database for an Enterprise.
- Apply Knowledge of XML Database, Web Database, Active Database and Temporal Database forMaintaining Data of an Enterprise.
- Model a Data Warehouse and Integration of a Data Mining System with A Data Warehouse.

REFERENCE BOOKS:

1. Ramez Elmasri, Shamkant B. Navathe, Fundamentals of Database Systems, Seventh Edition,
   Pearson Education, 2016.
2. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Database System Concepts, Seventh Edition,McGraw Hill Education 2020.
3. Brad Dayley, "Teach Yourself NoSQL with MongoDB in 24 Hours", Sams Publishing, 2014
4. Jiawei Han, Jian Pei and Hanghang Tong, Data Mining Concepts and Techniques, Fourth Edition,Morgan Kaufmann Publishers, 2022.

DATA STRUCTURES AND ALGORITHMS

TABLE OF CONTENTS

UNIT 1
RELATIONAL MODEL

CONTENTS

1.1 Introduction

A database acts like a large digital box for storing vast amount of information, such as details about people or events. Before building a database, a plan or blueprint, known as the "Entity-Relationship Model," is created. Using this model, the database is constructed with tables to hold different pieces of information; this step is known as the "Relational Model." To add, remove, or change information in the database, a special language named SQL (Structured Query Language) is used. Relational algebra comprises the rules and operations that explain how databases work and interact, providing the base for how SQL interacts with the database. To maintain an organized database and avoid unnecessary repetition or disorder, "Normalization" is applied. This method follows specific rules, known as "Normal Forms," to ensure everything is systematically arranged. In conclusion, these methods and rules work together to store and organize information effectively in a database, making interaction straightforward and keeping everything orderly and correct.

1.2 Learning Objectives

- Understand Entity-Relationship Model
- Understand Relational Data Model
- Covert ER Model to Relational Model
- Master Query Language
- Learn Relational Algebra
- Understand Data Normalization
- Study Structured Database

1.3 Overview

The Entity-Relationship (ER) Model and the Relational Data Model are foundational concepts in database design, focusing on representing and organizing data systematically. The ER Model utilizes entities, relationships, and attributes to map real-world objects, while the Relational Model structures data into tables, defining relations among them. Mapping from the ER Model to the Relational Model is a critical step in database design, ensuring logical structuring of data. Relational Algebra provides a theoretical foundation for processing and querying data in relational databases. A Query Language, like SQL, is used to interact with databases, allowing data retrieval, update, insertion, and deletion.

Database Normalization is a multi-step process aimed at efficiently organizing a database, reducing data redundancy and enhancing data integrity. The normal forms, including First (1NF), Second (2NF), Third (3NF), Boyce-Codd Normal Form (BCNF), and Fifth Normal Form (5NF), represent progressive stages in normalization. Each normal form addresses different aspects of redundancy and dependency, with the goal of optimizing the database structure. Understanding these models, the various normalization stages, and query languages is pivotal for designing robust, efficient, and well-structured databases, enabling effective data management and retrieval.

1.4 Database Model and Normalization

1.4.1 Introduction

Overview of Database

A database is a structured and organized collection of data designed to efficiently store, manage, and retrieve information. It serves as a digital repository that can vary in size from a small, localized system to a massive, distributed network, depending on the scale and requirements of the data being managed. Databases play an integral role in modern information systems, enabling organizations to store vast amounts of data, analyze it, and extract valuable insights for decision-making.

Key aspects of databases include:

1. **Data Structure**: Databases organize data into tables, consisting of rows and columns in the case of relational databases. Each table represents a specific entity or concept, and the relationships between tables allow for complex data modeling.
2. **Data Retrieval**: Databases provide mechanisms for users and applications to query and retrieve data efficiently. Structured Query Language (SQL) is commonly used for this purpose in relational databases.
3. **Data Integrity:** Databases enforce rules and constraints to ensure data accuracy and consistency. This includes primary keys, foreign keys, and unique constraints.
4. **Concurrency Control:** Databases support multiple users or applications accessing data simultaneously while preventing conflicts and ensuring data integrity.
5. **Security:** Database systems offer authentication and authorization mechanisms to protect sensitive data from unauthorized access.
6. **Scalability:** Databases can scale horizontally or vertically to handle increasing data volumes and user loads.
7. **Data Recovery:** Backup and recovery features safeguard against data loss due to system failures or human errors.

There are various types of databases, including:

1. **Relational Databases:** Such as MySQL, Oracle, and PostgreSQL, which organize data into structured tables with well-defined relationships.
2. **NoSQL Databases:** Like MongoDB and Cassandra, which are designed for unstructured or semi-structured data and offer flexibility in data modeling.
3. **Graph Databases:** Suited for managing complex, interconnected data structures, often used in social networks and recommendation systems.
4. **Document Databases:** Ideal for storing hierarchical, document-based data like JSON or XML.
5. **Key-Value Stores:** Efficiently manage simple data with key-value pairs.
6. **Column-Family Stores:** Designed for high-write, large-scale data handling.

Databases serve as the backbone of applications, providing support for a wide range of systems, including e-commerce websites to healthcare systems and scientific research, making them indispensable in the digital age. Effective database design and management are critical for data accuracy, security, and the success of information-driven businesses and applications. The key components that constitute a database system include users, applications, DBMS, data, host system depicted in Figure 1.1.

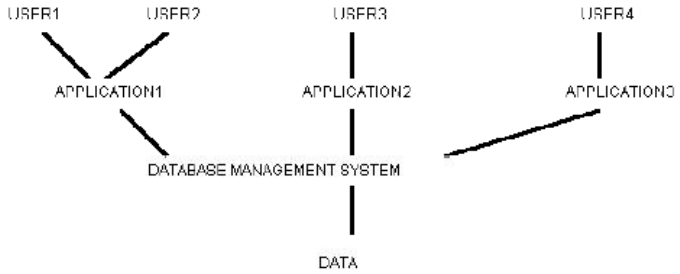


Figure. 1.1 Key Components of Database System

Data Description in a DBMS

Metadata is essentially "data about data." It describes the structure of databases, including tables, attributes, and relationships. Metadata helps the system to understand the organization of data and the methods to access it. The "Data Dictionary" acts as a repository for this metadata, providing comprehensive details about every element within the database such as names, data types, and constraints. The "Schema" outlines the logical structure of the entire database including definitions of tables, fields, relationships, views, indexes, packages, procedures, functions, queues, triggers, types, sequences, materialized views, synonyms, database links, directories, XML schemas, and more. Sub-schema depicts a subset of the main schema, offering a way to display pertinent data to specific users, while concealing other data from them.

Data Storage in a DBMS

Database systems store information in specialized computer files, resembling organized shelves with data neatly arranged in boxes. Sometimes, these boxes can be grouped into areas called as "table spaces", akin sections in a library. When accessing information in a database, it's presented in rows, similar to lines in a book. To expedite data retrieval, databases employ 'indexes' as shortcuts, although these can occasionally affect the speed of adding new data. s time goes on, with data additions and removals, the 'shelves' can become disorganized. Periodic tidying is necessary for smooth operations. Sometimes, the same information is stored in multiple places, which is inefficient. There's a technique called "normalization" to avoid this, much like decluttering a room. Lastly, Finally, creating backup copies of our data is crucial in case of mishaps, much like backing up our photos to prevent loss.

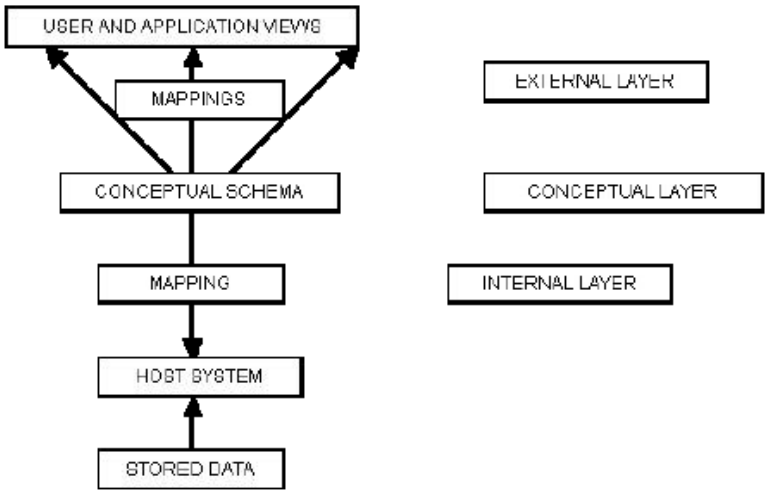


Figure 1.2. Layering of a Database Management System

Database Management Systems (DBMS) are typically organized into layers for efficient data management. The core components include the physical layer, responsible for data storage and retrieval; the logical layer, which manages schema design and query optimization; and the user interface layer, which offers an interface for users to interact with the database, as illustrated in Figure 1.2.

1.4.2 Introduction to Database Design

Database design is the intricate process of structuring and organizing data for efficient storage and retrieval. It forms the foundation for a robust data management system, enabling smooth and efficient operations. A primary tool utilized in this process is the Entity-Relationship (ER) diagram, which visually represents the interrelationships between data pieces.

Three Steps of Database Design:

Conceptual Design:

Purpose: At this initial stage, designers establish the high-level structure of the database by identifying the main entities, their attributes, and their relationships without focusing on specific details.

ER Diagram Role: ER diagrams play a pivotal role. Designers draw entities as rectangles, attributes as ovals connected to respective entities, and relationships as diamonds connecting related entities. For instance, in a school system, entities might be "Student" and "Class" with a relationship "Enrolls In" connecting them.

Logical Design:

Purpose: This phase refines the conceptual model into a more detailed structure, considering specific requirements and constraints. It involves creating tables, keys, and indices.

ER Diagram Role: In this phase, ER diagrams are expanded, mapping entity sets to tables and indicating relationships with foreign keys. Data normalization, which aims to reduce data redundancy, is a primary focus during this step.

Physical Design:

Purpose: The logical design is translated into physical structures. Decisions about data storage, access paths, and optimization methods are made to ensure efficient data access and storage.
ER Diagram Role: While ER diagrams are less directly applicable in this step, the structured blueprint from the earlier stages guides the physical implementation, ensuring that the real-world deployment of the database aligns with the intended design.

Database design, aided by ER diagrams, ensures structured, efficient, and adaptable database. The three-step process—conceptual, logical, and physical—ensures the design meets user requirements while optimizes for performance and scalability.

Entities, Relationship and Attributes

The fundamental concepts of the E-R model include entity sets, relationship sets, and attributes. Entities are unique real-world items or objects. In a university, individuals are examples of entities, each possessing distinct characteristics. For instance, a person may have a unique ID like '677-89-9011,' providing a specific identifier. Similarly, courses are also entities, with each having a unique course ID. Entities can be tangible, such as a person or a book, or more abstract, like a particular course session or flight booking.

Entity Sets: An entity set groups similar entities that share common attributes. For instance, all university instructors can belong to the "instructor" entity set. Each entity in this set has attributes, like ID and name. The specific group of entities in an entity set at a given time is its extension. So, the actual instructors in a university are the extension of the "instructor" entity set. Notably, entity sets can overlap; a "person" at a university could belong to both "student" and "instructor" sets. Every entity has attributes, which are descriptive features. An instructor might have attributes like ID, name, department, and salary. A university database will contain multiple entity sets, such as "instructor", "student", and "course", each with specific attributes. Figure 1.3 shows the attributes of a student entity set.

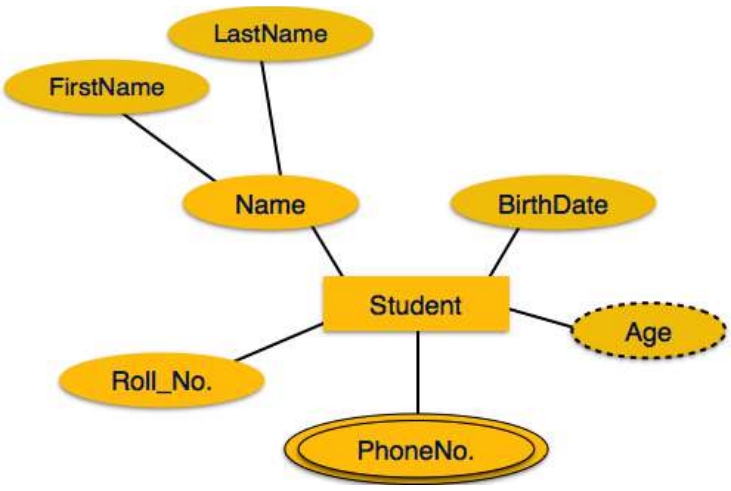


Figure 1.3. Attributes for a Student Entity in the University

Entities are depicted as rectangles, and each rectangle is labeled with the name of the entity set it represents. Attributes, describing entity properties, are illustrated as ellipses each linked directly to its respective entity (rectangle). Composite attributes with multiple components are displayed in a tree-like structure branching from the main ellipse. Multivalued attributes are denoted by a double ellipse, and derived attributes are indicated with a dashed ellipse.

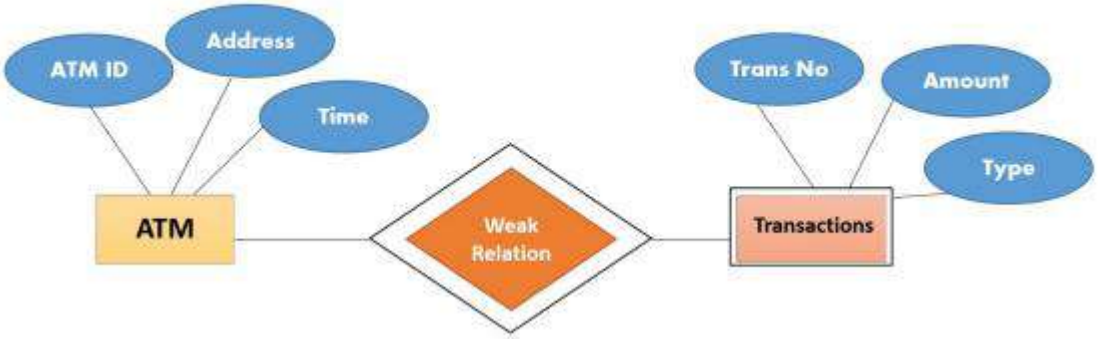
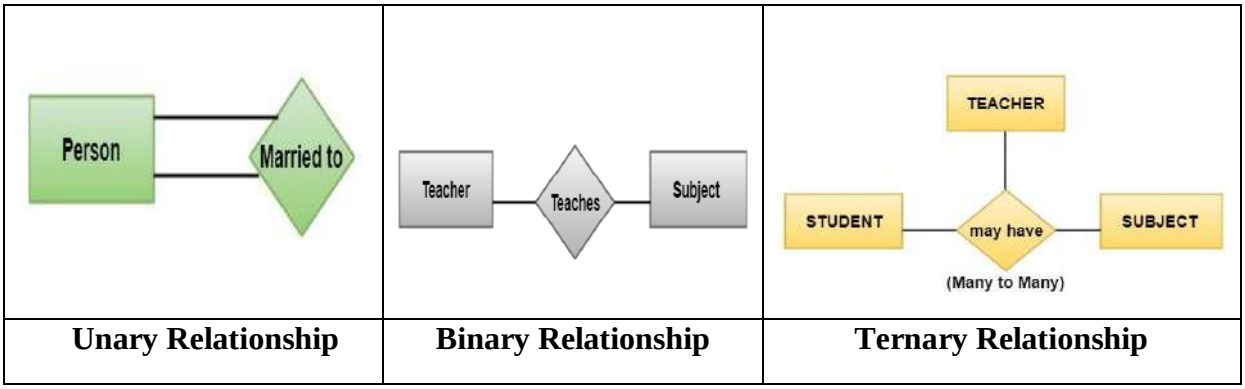


Figure 1.4 Illustration of the Relationship Between Weak and Strong Entities

A Weak Entity Set lacks primary attributes to uniquely identify its entities independently. Instead, it relies on a specific combination of attributes and its relationship with a stronger, primary entity set. For example, consider a 'Transaction' entity representing a weak relationship, like an ATM withdrawal. While a transaction may have identifiers like 'Trans No,' 'Amount,' and 'Type,' its unique determination depends on the specific ATM involved. In contrast, a Strong Entity Set possesses a primary key, ensuring that its entities are uniquely identifiable without relying on another entity set. In ER diagrams depicted in Figure 1.4, weak entity sets are represented by a double rectangle, and their identifying relationships are illustrated with a double diamond. Conversely, strong entity sets are shown using a single rectangle.

Relationship: Relationships are represented by a diamond-shaped boxes containing the relationship name. Entities, represented by rectangles connect to this diamond with lines.There are three primary types of relationships: unary, where an entity is related to itself; binary, connecting two distinct entities; and ternary, linking three separate entities together. Each type serves specific purposes and carries implications in database design as given in the Figure 1.5.



Unary Relationship	Binary Relationship	Ternary Relationship

Figure 1.5 Types of Relationships

In the ER diagram, key constraints determine how entities in a relationship can be uniquely identified and associated with entities in another set.

- Super key: A set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. E.g. Student ID, Email, Student ID, First name etc.
- Candidate Key: A super key such that no proper subset of it is also a super key. Student ID, Email.
- Primary Key: One of the candidate keys chosen by the database designer to uniquely identify the entity set Student ID.
- Foreign Key: An attribute or set of attributes in a relation that allows for the identification of a unique tuple in another relation. E.g Student ID in the Student table is the primary key. Enrollment ID in the Enrollment table is the primary key. However, the Enrollment table also has the Student ID attribute, which refers to the Student ID in the Student table. In this context, the Student ID in the Enrollment table acts as a foreign key.

In ER diagrams, key constraints are crucial for the following:

Participation Constraint: Dictates how many instances of an entity set can participate in a relationship. It can be total (every entity must participate) or partial (not every entity needs to participate).

Cardinality Constraint: Defines the numeric relationship between the instances of two entity sets. Examples include 1:1 (one-to-one), 1:M (one-to-many), M:1 (many-to-one), and M:N (many-to-many).

These constraints are essential or maintaining data integrity in the database, preventing errors and inconsistencies in data storage and retrieval.

A binary relationship involves two participating entities. Cardinality refers the number of instances of an entity from a relation that can be linked with the relationship.

One-to-one (1:1): This occurs when only one instance of each entity is associated with the relationship. The Figure 1.6 illustrates this one-to-one association. e.g. Each student has one registration number, and each registration number belongs to one student.

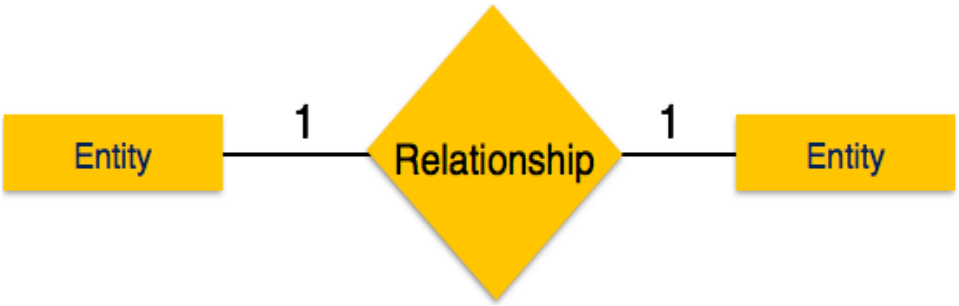


Figure 1.6. One-to-One Relationship

One-to-many (1:N): In this type, one instance of the entity on the left can be linked with multiple instances of the entity on the right. The Figure 1.7 showcases this one-to-many relationship. E.g. One teacher can teach many courses, but each course has one teacher

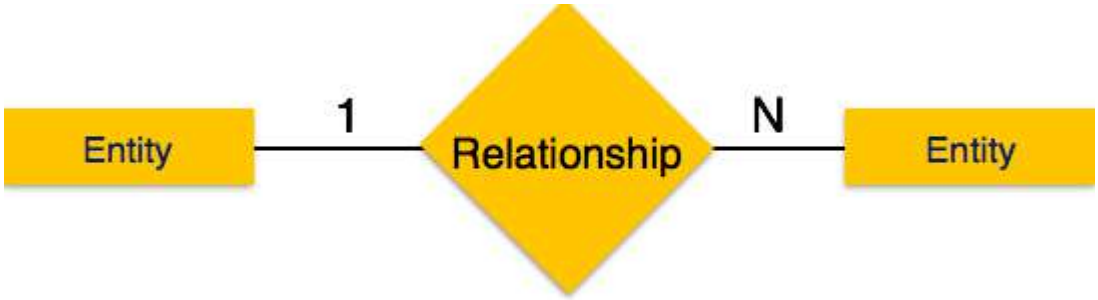


Figure 1.7. One-to-Many Relationship

Many-to-one (N:1): Here, multiple instances of the entity on the left can be linked with a single instance of the entity on the right. The image provided in Figure 1.8 represents this many-to-one association. E.g. Many students attend one school; each school has multiple students. E.g. Many books can be published by one publisher, but each book has one publisher.

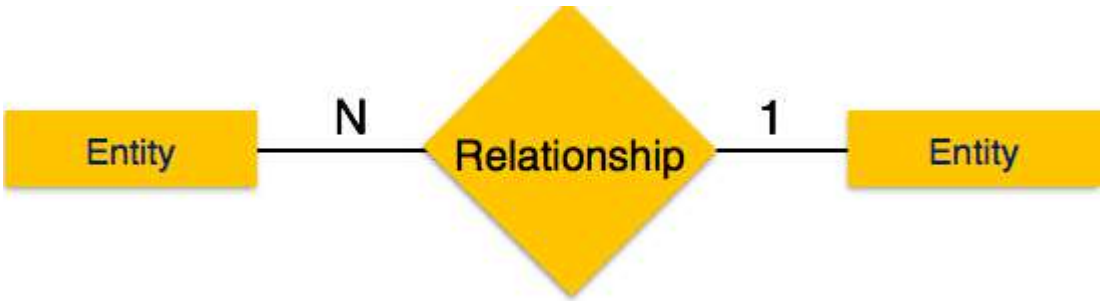


Figure 1.8 Many-to-One Relationship

Many-to-many: This relationship allows multiple instances from both participating entities to be interrelated. The given image in Figure 1.9 portrays this many-to-many linkage. E.g. One student can enroll in many subjects, and each subject can have many students.

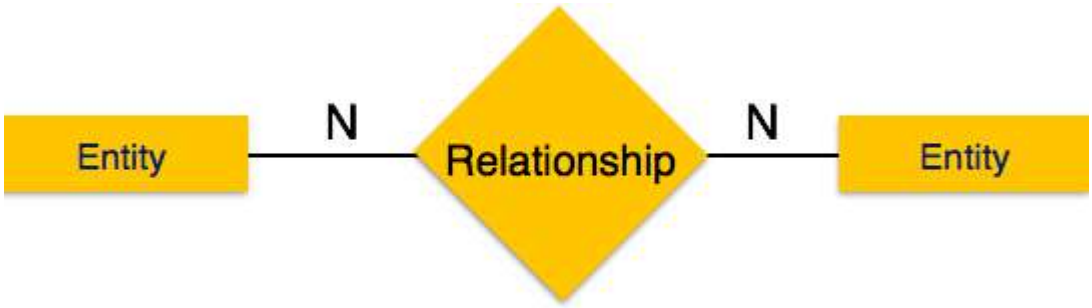


Figure 1.9. Many-to-Many Relationship

Participation Constraint

Participation constraint in database design determines whether an entity must or can participate in a relationship, as shown in Figure 1.10. It can be total (mandatory) represented by double lines, where every entity is involved in the relationship, or partial (optional), represented by single lines, where not all entities are involved in the relationship.

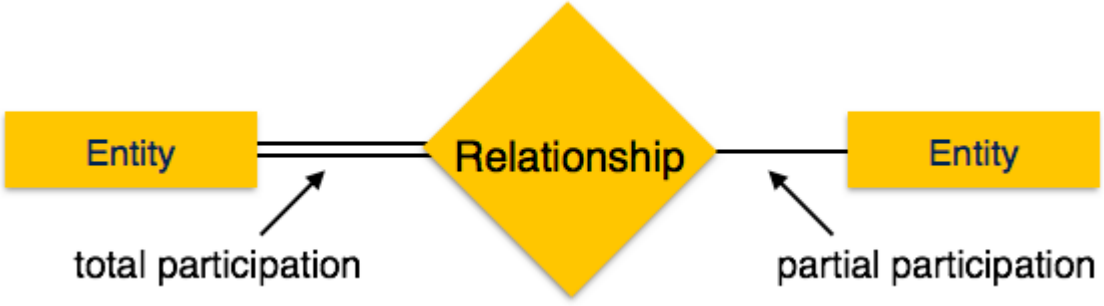


Figure 1.10. Participation Constraints

For instance, in total participation, every book must have at least one author, and every author must have written at least one book. This represents total participation. In contrast, in partial participation, not every publisher has to publish a book. Some books might not have a publisher yet, and some publishers might not have published any books. This illustrates partial participation.

The ER Model facilitates the hierarchical representation of database entities, incorporating both generalization and specialization. Generalization involves grouping entities to create a more generalized view. For example, individual students like Mira are generalized as "students," and students are further generalized as "persons". In contract, specialization represents a reverse process, where a person is specialized into a student, with Mira as an example. Generalization involves combining entities based on their shared characteristics to form a more comprehensive entity. For example, entities such as pigeons, house sparrows, crows, and doves can all be grouped under the category of 'Birds,' as illustrated in Figure 1.11.

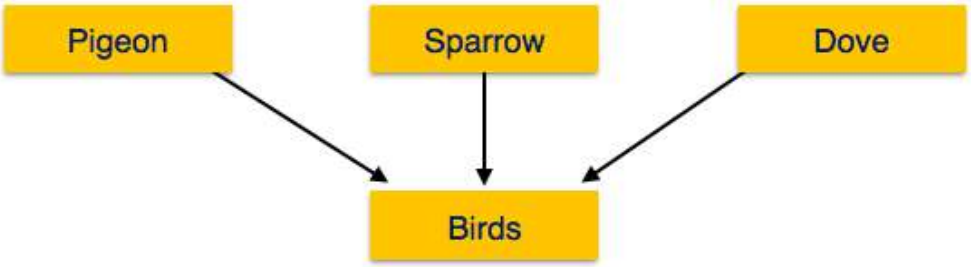


Figure 1.11. Generalization Example

Specialization, unlike generalization, involves dividing a set of entities into sub-groups based on their unique characteristics. For instance, the broader category 'Person,' includes common attributes like name, date of birth, and gender. However, within a company, individuals can be categorized as employees, employers, customers, or vendors, depending on their roles. Similarly, in the context of a school database, individuals can be specialized as teachers, students, or staff members based on their respective roles and functions within the school, as depicted in Figure 1.12.

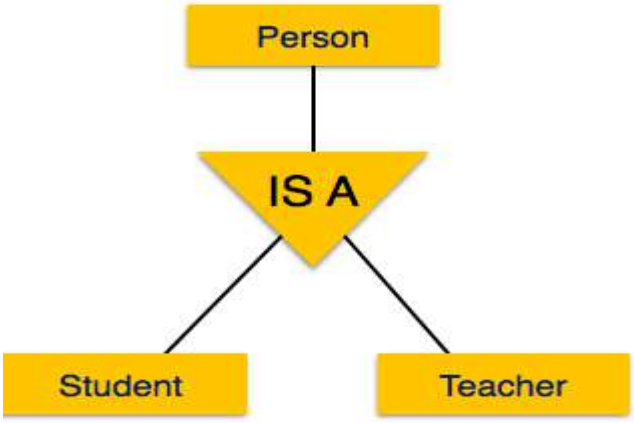


Figure 1.12. Specialization Example

Inheritance, a significant aspect of Generalization and Specialization, permits lower-level entities to acquire the attributes of higher-level entities. This feature, along with other ER-Model components, is employed to build object classes in object-oriented programming. It often involves concealing the intricate details of entities from users, a concept known as abstraction. For instance, lower-level entities like 'Student' or 'Teacher' can inherit attributes such as name, age, and gender from the higher-level 'Person' class, as illustrated in Figure 1.13.

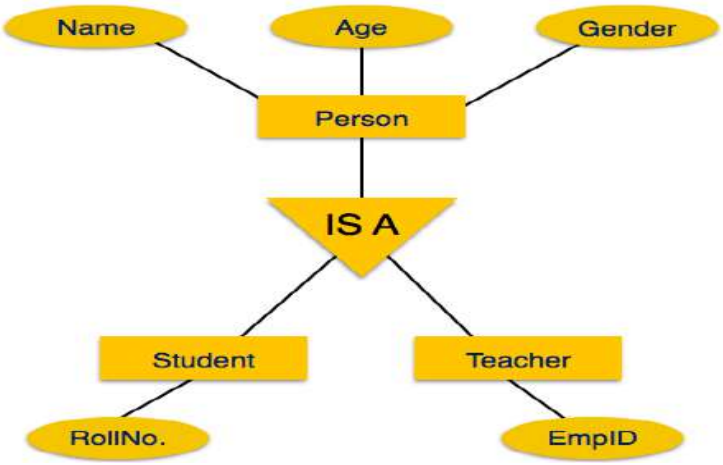


Figure 1.13. Inheritance Example

Database design is an ongoing process that aligns with an organization's evolving needs. As business requirements evolve, so do the data storage demands. In the early phases of application development, database designers often recognize the need for modifications at the conceptual, logical, or physical schema levels, affecting the entire database application. A robust database design aims to anticipate organization's future needs, striving to keep schema modifications minimal. It's important to distinguish between permanent constraints, like unique instructor IDs, and those subject to change, such as department affiliations due to policy adjustments. Effective designs account for potential alterations and consider data exchange between different schemas, addressing real-world challenges. Furthermore, successful database design involves extensive interaction with domain experts and end-users, to ensure the successful implementation of data systems within an enterprise.

1.4.3 Relational Data Model

The Relational Data Model is a foundational concept in database management. It's a way of organizing data in a tabular form, where data is structured into tables with rows and columns. In this model:

1. **Tables (Relations):** Data is organized into tables, also called relations. Each table represents a specific entity or concept, and each row in the table represents a unique instance of that

entity. For example, a table might represent "Customers," and each row in that table represents a single customer.

2. **Columns (Attributes)**: Each table has columns, also known as attributes. Columns define the type of data that can be stored in them. For instance, in a "Customers" table, you might have columns like "Customer ID," "First Name," "Last Name," and "Email."

3. **Keys:** A key is a column or set of columns that uniquely identifies each row in a table. The Primary Key is the main identifier for a table, ensuring that each row is unique. There are also Foreign Keys, which link tables together based on common attributes.

4. **Relationships**: Relational databases allow you to establish relationships between tables. For example, you can link a "Orders" table to a "Customers" table using the Customer ID as a Foreign Key to associate each order with a customer.

5. **Data Integrity:** Relational databases enforce data integrity through constraints like Primary Keys, Foreign Keys, and other rules to maintain the accuracy and consistency of data.

6. **SQL (Structured Query Language)**: To interact with relational databases, you use SQL, a domain-specific language for querying and manipulating data. SQL allows you to perform operations like inserting, updating, deleting, and retrieving data from the database.

The Relational Data Model provides a flexible and efficient way to store, retrieve, and manage structured data, making it one of the most widely used database models in the world. Popular relational database management systems (RDBMS) include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

Illustrate the relational model with an example of a simple database system for a library, which consists of four tables:

Authors (author ID, First Name, Last Name, Date of Birth)

Books (ISBN, Title, Published Date, Author ID)

Borrowers (Borrower ID, First Name, Last Name, Membership Date)

Loans (Loan ID, Borrower ID, ISBN, Loan Date, Return Date)

The **Authors** table lists all the authors, each with a unique Author ID, their first and last names, and their date of birth. The **Books** table enumerates all the books in the library, with each book having a unique `ISBN`, a title, a publication date, and an `Author ID` that references the Authors table to denote the book's author. The **Borrowers** table records all individuals who borrow books from the library. Every borrower is uniquely identified by a `Borrower ID`, and the table also

includes their first and last names, as well as their membership initiation date. The **Loans** table documents the borrowing history, noting which borrowers have taken out which books. Each borrowing instance has a unique `Loan ID`, the `Borrower ID` of the individual who borrowed the book, the `ISBN` of the borrowed book, the date of borrowing, and the return date.

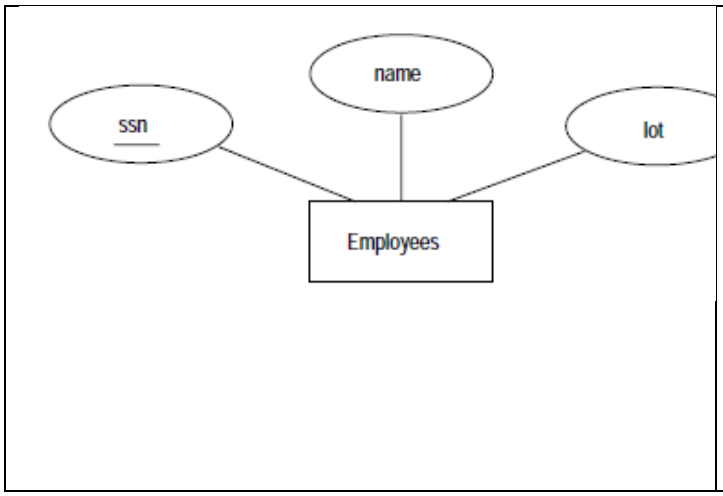
This relational model empowers the library system to systematically track books, authors, borrowers, and borrowing transactions. By leveraging joins, one can query across tables, enabling answers to questions like "Which books has a particular borrower checked out?" or "Which books were authored by a specific individual?".

1.4.4 Mapping Entity Relationship Model to Relational Model

The Entity-Relationship (ER) model provides a high-level graphical representation of a database's design. After creating an ER diagram that describes a database, the next step is to converting it into a relational database schema. This transformation includes the creation of tables and associated constraints based on the the ER design. The process involves converting entities and relationships in the ER diagram into tables, columns, and constraints in the relational schema. Next transformation is an ER diagram into a series of tables that make up a relational database schema.

- **Entity set to Tables**

  Converting an entity set to a relation is a direct process: Every attribute in the entity set translates into a column in the corresponding table. It's important to mention that the domain of each attribute, as well as the primary key of the entity set, is well-defined. For the "Employees" entity set, which has the attributes: ssn, name, and lot, as depicted in Figure 1.14 A. An instance of the "Employees" entity set, which includes three distinct employee entities, can be observed in Figure 1.14 B, presented in a table format.



ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

A : The Employees Entity Set	B : An Instance of the Employees Entity Set

Figure 1.14

CREATE TABLE Employees (ssn CHAR(11) PRIMARY KEY, name CHAR(30), lot INTEGER);

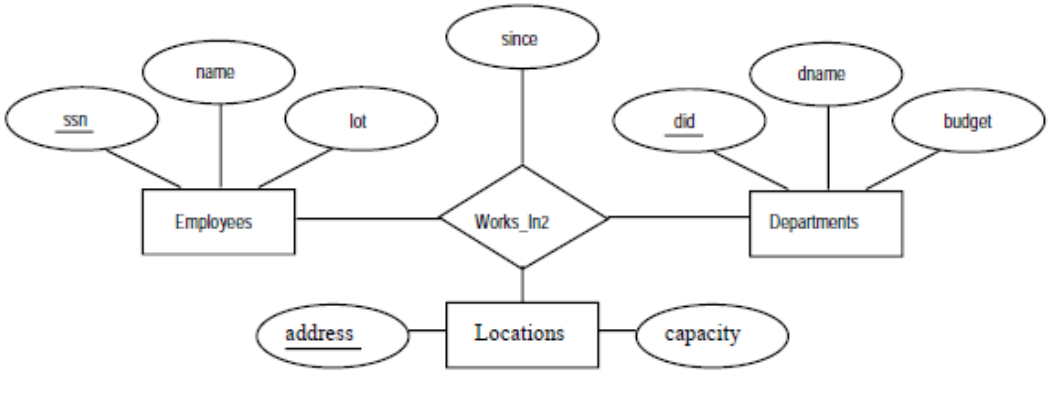
The SQL statement encapsulates the aforementioned details, encompassing both the domain constraints and key information.

- **Relationship Sets(without Constraints) to Tables**

  To translate a relationship set into a relation in a relational database model, especially when there are no key or participation constraints, certain steps are followed. The objective is to ensure that every entity involved in the relationship is distinctly recognized and that any attributes describing the relationship are captured.

When transitioning a relationship set into a relation:

1. The primary key attributes from every participating entity set are used. These serve as foreign key fields in the relation.
2. Attributes that describe the relationship set itself are included.
3. All attributes not directly describing the relationship function as a superkey for the resulting relation. In the absence of key constraints, this superkey can also be considered a candidate key.



A : A Ternary Relationship Set

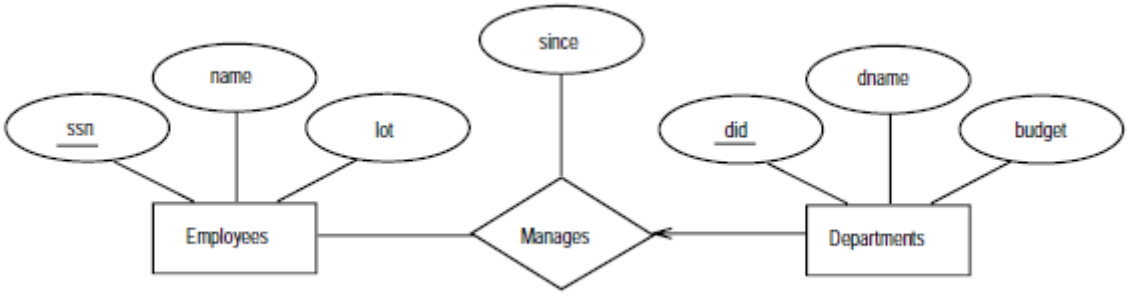
```
CREATE TABLE Works_In2 ( ssn      CHAR(11),
                         did      INTEGER,
                         address  CHAR(20),
                         since    DATE,
                         PRIMARY KEY (ssn, did, address),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                         FOREIGN KEY (address) REFERENCES Locations,
                         FOREIGN KEY (did) REFERENCES Departments )
```

B : SQL Definition

Figure 1.15

As illustrated in the figure 1.15 the fields 'address', 'did', and 'ssn' cannot have null values. Since these fields form the primary key for 'Works In2', there's an implicit NOT NULL constraint for each of them.

- **Translating Relationship Sets with Key Constraints**
  In Figure 1.16, the 'Manages' relationship set has attributes: ssn, did, and since. Since each department has only one manager, 'did' is unique and serves as the key for 'Manages'.Using key constraints in translating relationship sets can prevent the need for separate tables. For instance, embedding the 'Manages' relationship directly within the 'Departments' table simplifies queries but may lead to wasted space with null values when managers are absent.



A : Key Constraint on Manages

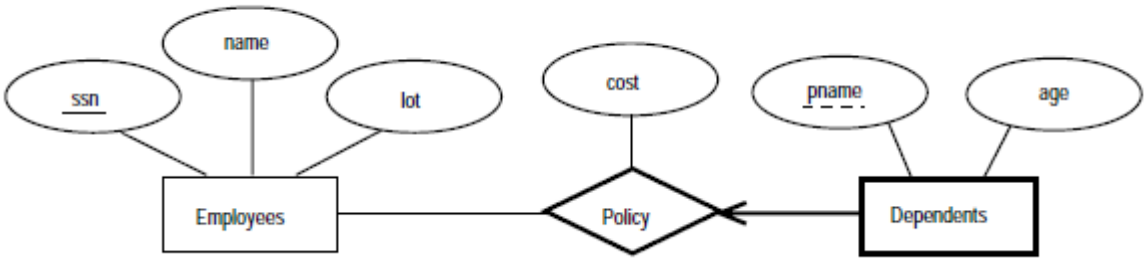
```
CREATE TABLE Manages ( ssn       CHAR(11),
                       did       INTEGER,
                       since     DATE,
                       PRIMARY KEY (did),
                       FOREIGN KEY (ssn) REFERENCES Employees,
                       FOREIGN KEY (did) REFERENCES Departments )
```

B : SQL Definition

Figure 1.16

- **Translating Weak Entity Sets**

  A weak entity set consistently participates in a one-to-many binary relationship and is characterized by both a key constraint and total participation, as shown in Figure 1.17. Such a weak entity possesses only a partial key. When an owning entity, such as "Employees," is deleted, its linked weak entities are concurrently removed. To uniquely identify a "Dependents" entity, it's essential to consider both its partial key, "p name," and the "Employees" key.



A : The Dependents Weak Entity Set

```
CREATE TABLE Dep Policy ( p name CHAR(20),
                          age INTEGER,
                          cost REAL,
                          PRIMARY KEY (pname, ssn),
                          FOREIGN KEY (ssn) REFERENCES Employees
                          ON DELETE CASCADE )
```

B : SQL Definition
Figure 1.17

The primary key for "Dependents" consists of "pname" and "ssn". It's imperative to link every "Dependents" entity to an "Employees" entity, making sure "ssn" isn't null since it's a component of the primary key. By applying the CASCADE option, deleting an "Employees" tuple will also delete its associated data.

1.4.5 Relational Algebra

In 1970, Edgar F. Codd, often referred to as the Father of Database Management Systems (DBMS), introduced Relational Algebra. This concept is alternatively known as Procedural Query Language (PQL) because in PQ, programmers or users are required to define two essential aspects: "What needs to be done" and "How it should be done."

Relational Algebra is the tool for retrieving data from a database. The first step involves specifying the data retrieval task in the query, denoted as 'What to Do.' It's also necessary to articulate the method or procedure within the query for accessing data from the database, described as 'How to Do' or the approach for data retrieval.

Fundamental Operations

The types of relational operations are depicted in Figure 1.18.a. The basic unary operations are select, project, and rename because they operate on a single relation. In contrast, the remaining three operations—Union, set difference, and Cartesian product—involve working with pairs of relations and are therefore referred to as binary operations and the sample for all the operations is depicted in Figure 1.18.b.

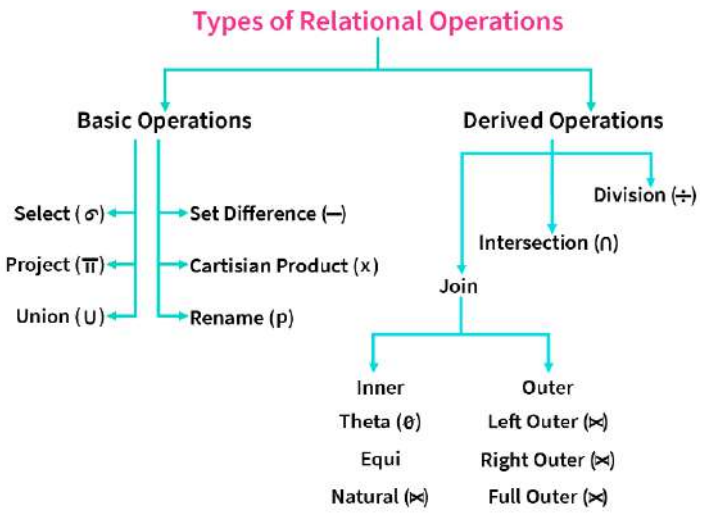


Figure 1.18.a. Types of Relational Operations

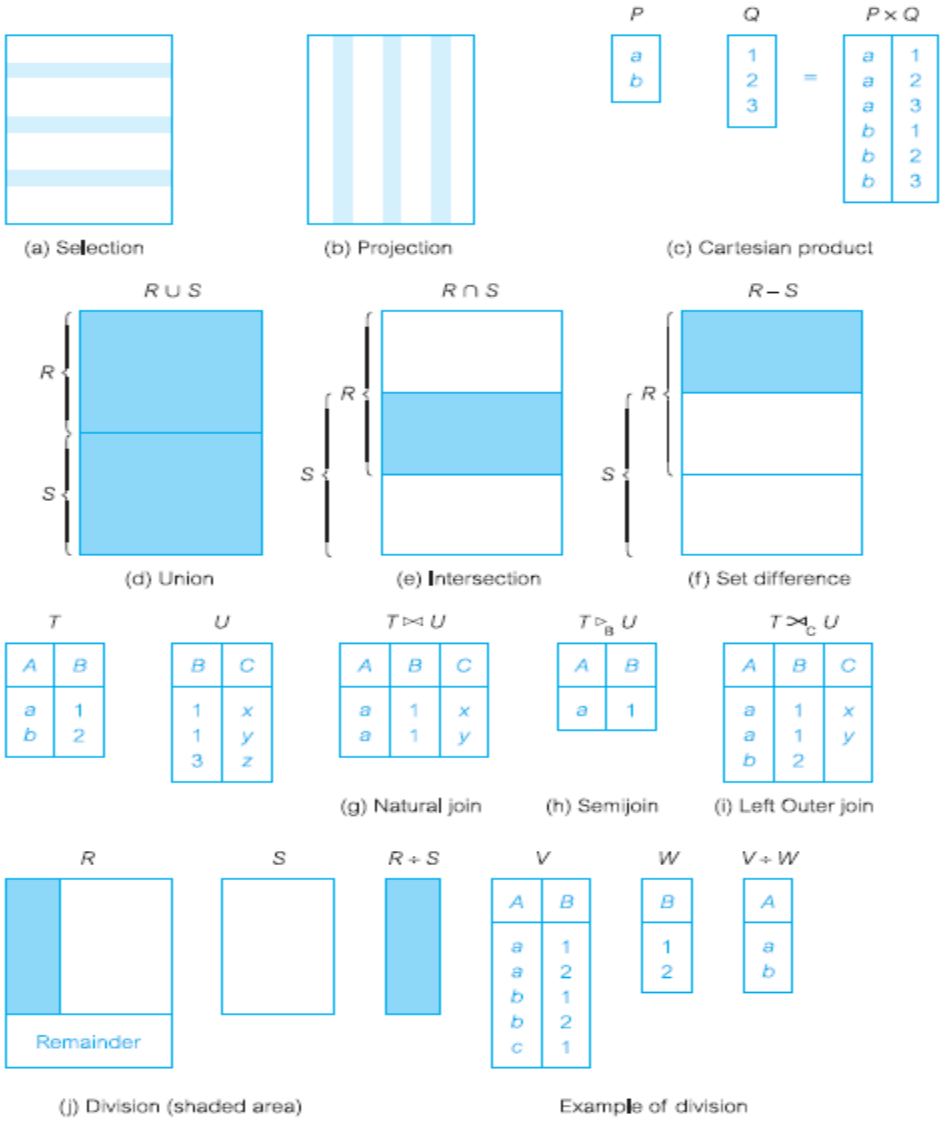


Fig 1.18.b. Illustration showing the function of the relational algebra operations

Selection Operation (or Restriction)

σ p(R): The Selection operation works on a single relation R and defines a relation that contains only those tuples of R that satisfy the specified condition (predicate).

Example (Table 1.1)

- List all staff with a salary greater than 10,000. **$\sigma_{salary > 10,000}$(Staff)**

Table 1.1: Staff

staffNo	fName	lName	position	sex	DOB	salary	branchN
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

$\sigma_{salary> 10,000}$(Staff) - Selecting salary > 10000 from the Staff relation.

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

Projection Operation

∏ a(r): The Projection operation works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.

Example (Table 1.1)

- Produce a list of salaries for all staff, showing only the staff No, fName, lName, and salary details.

∏ staff No, f Name, l Name,Salary(Staff)

staffNo	fName	lName	salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SA9	Mary	Howe	9000
SG5	Susan	Brand	24000
SL41	Julie	Lee	9000

Projecting the Staff relation over the staffNo, fName, lName, and salary attributes.

Set Operations: The Selection and Projection operations extract information from only one relation. There are obviously cases where we would like to combine information from several relations. In the remainder of this section, the binary operations of the relational algebra, starting with the set operations of Union, Set difference, Intersection, and Cartesian product are explored with Depositor (R) and Borrow (S) relations.

DEPOSITOR RELATION (R)

CUSTOMER_ NAME	ACCOUNT_ NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

BORROW RELATION(S)

CUSTOMER_ NAME	LOAN_ NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

Union:

R∪S : The union of two relations R and S defines a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible.

If R and S have I and J tuples, respectively, their union is obtained by concatenating them into one relation with a maximum of (I + J) tuples. Union is possible only if the schemas of the two relations match, that is, if they have the same number of attributes with each pair of corresponding attributes having the same domain.

Set difference:
R − S : The Set difference operation defines a relation consisting of the tuples that are in relation R, but not in S. R and S must be union-compatible.

Intersection :
R ∩ S : The Intersection operation defines a relation consisting of the set of all tuples that are in both R and S. R and S must be union-compatible.

<table>
<tr><td colspan="4">Query 1 : ∏ CUSTOMER_NAME (BORROW) ∪ ∏ CUSTOMER_NAME (DEPOSITOR)</td></tr>
<tr><td>OUTPUT</td><td>CUSTOMER_NAME
Johnson
Smith
Hayes
Turner</td><td>Jones
Lindsay
Jackson
Curry</td><td>Williams
Mayes</td></tr>
<tr><td colspan="4">Query 2 : ∏ CUSTOMER_NAME (BORROW) ∩ ∏ CUSTOMER_NAME (DEPOSITOR)</td></tr>
<tr><td colspan="2">OUTPUT</td><td colspan="2">CUSTOMER_NAME
Smith
Jones</td></tr>
<tr><td colspan="4">Query 3 : ∏ CUSTOMER_NAME (BORROW) - ∏ CUSTOMER_NAME (DEPOSITOR)</td></tr>
<tr><td colspan="2">OUTPUT</td><td colspan="2">CUSTOMER_NAME
Jackson
Hayes
Willians
Curry</td></tr>
</table>

Cartesian product: R × S The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S. The Cartesian product operation multiplies two relations to define another relation consisting of all possible pairs of

tuples from the two relations. Therefore, if one relation has I tuples and N attributes and the other has J tuples and M attributes, the Cartesian product relation will contain (I * J) tuples with (N + M) attributes. It is possible that the two relations may have attributes with the same name. In this case, the attribute names are prefixed with the relation name to maintain the uniqueness of attribute names within a relation.

EMPLOYEE

EMPID	EMPNAME	EMPDEPT
1	Smith	A
2	Harry	C
3	John	B

DEPARTMENT

DEPTNO	DEPTNAME
A	Marketing
B	Sales
C	Legal

Query : EMPLOYEE X DEPARTMENT

OUTPUT

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

Join Operations

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

EMPLOYEE		SALARY		RESULT		
EMP_CODE	**EMP_NAME**	**EMP_CODE**	**SALARY**	Query: (EMPLOYEE ⋈SALARY)		
				EMP_CODE	**EMP_NAME**	**SALARY**
101	Stephan	101	50000	101	Stephan	50000
102	Jack	102	30000	102	Jack	30000
103	Harry	103	25000	103	Harry	25000

Division : The division operator results in columns values in one table for which there are other matching column values corresponding to every row in another table.

k	x	y
1	A	2
1	B	4
2	A	2
3	B	4
4	B	4
3	A	2

Table A

x	y
A	2
B	4

Table B

k
1
3

A DIV B

1.4.6 Structured Query Language

In a relational database, all data is stored in simple two-dimensional tables known as relations. In Table 1.2, shows an example of a relation that stores data regarding a number of employees in a firm.

Table 1.2 A Relation

Example: Employee Table - EMP			
EMP #	**EMP NAME**	**DEPT NAME**	**GRADE**
1	F Jones	SALES	6
2	P Smith	ACCOUNTS	6
3	K Chan	SALES	4
6	J Peters	SALES	5
9	S Abdul	ACCOUNTS	3

All items sharing similar characteristics are organized within the same relation. These characteristics are defined by the column headings (such as EMPNO, EMPNAME, DEPTNAME, GRADE) at the top of the table. Each occurrence or instance of a specific item (in this case, EMP) is represented by rows of data beneath these column headings. Interpreting each row is straightforward; for instance, the first row describes an EMP with an EMPNO of 1, EMPNAME as P. Jones, DEPTNAME as SALES, and a GRADE of 6.

A tuple is an ordered list of values, and the meaning of each value is determined by its position in the tuple. For example, in the first tuple, the first value (1) represents the EMPNO, the second value (P Jones) represents the EMPNAME, and so on. The number of tuples in a relation determines its cardinality. In this case, we have a relation with cardinality of five.

"SQL stands for "Structured Query Language." This language allows us to formulate complex queries for databases and serves as a means of creating databases. SQL is widely used in the database industry. Many database products support SQL, which means that if you learn how to use SQL, you can apply this knowledge to various database systems like MS Access, SQL Server, Oracle, Ingres, and many others.

SQL is designed for relational databases, which store data in tables (often called relations). A database is a collection of tables, each containing a list of records. Each record in a table has the same structure, consisting of a fixed number of "fields" of a specific data type.

SQL comprises several components:

a. Data Definition Language (DDL) for defining the database structure and controlling access to the data.
b. Data Manipulation Language (DML) for retrieving and updating data.
c. Embedded and dynamic SQL
d. Security
e. Transaction management
f. Client-server execution and remote database access.

Query Processing

The process of query execution involves several key stages. First, there's parsing, where a parser converts a user-submitted query, often written in a high-level language, into an expression using algebraic operators. Next, optimization becomes crucial in query processing design, as it takes the expression and constructs an efficient execution plan. This plan determines the order of operator execution and selects suitable algorithms for implementation. The next phase involves Code Generation for the Query, where planned code is generated to ensure high-performance retrieval of the query's output. Subsequently, Code Execution by the Database Processor takes place, managed by an execution engine module that delivers the query's result to the user by executing the designated query plan. Finally, the query's outcome is presented.

The SQL statement performs the 3 basic types of operation and they are shown in Figure 1.19

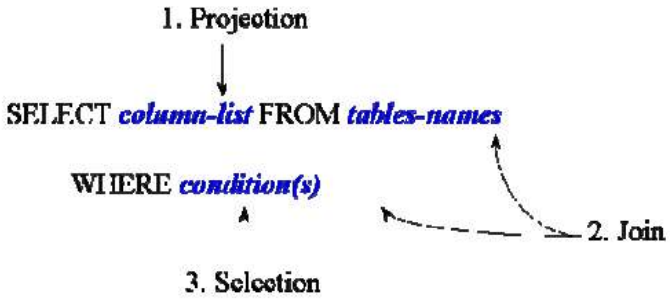


Figure 1.19. Basic of Types of SQL Operations

SQL CREATE TABLE

A Table is a combination of rows and columns. For creating a table, we have to define the structure of a table by adding names to columns and providing data type and size of data to be stored in columns.

*Here **table_name** is name of the table, **column** is the name of column*

SYNTAX	Example
CREATE table table_name (column1 datatype (size), column2 datatype (size), .	CREATE TABLE Customer(CustomerID INT PRIMARY KEY, CustomerNameVARCHAR(50), LastNameVARCHAR(50), Country VARCHAR(50),

. columnN datatype(size));	Age int(2), Phone int(10));

Output

CustomerID	CustomerName	LastName	Country	Age	Phone
empty					

- char(n): A fixed-length character string with user-specified length n.
- varchar(n): A variable-length character string with user-specified maximum length n.
- int: An integer (a finite subset of the integers that is machine dependent).
- primary key (Aj1 , Aj2, . . . , Ajm): The primary-key specification says that attributes Aj1, Aj2, . . . , Ajm form the primary key for the relation. The primarykey attributes are required to be nonnull and unique;
- **not null**: The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute

Adding Data to the Database (INSERT)

The INSERT INTO statement is used to insert new records in a table

SYNTAX	**EXAMPLE**	
INSERT INTO Table Name [(column List)] VALUES (data Value List)	INSERT INTO Staff VALUES (‘SG16’, ‘Alan’, ‘Brown’, ‘Assistant’, ‘M’, DATE ‘1957-05-25’, 8300, ‘B003’);	INSERT INTO Staff (staffNo, fName, lName, position, salary, branchNo) VALUES (‘SG44’, ‘Anne’, ‘Jones’, ‘Assistant’, 8100, ‘B003’);

Output

staffNo	fName	lName	Position	Sex	DOB	Salary	branchNo

SG16	**Alan**	**Brown**	**Assistant**	**M**	**1957-05-25**	**8300**	**B003**
SG44	**Anne**	**Jones**	**Assistant**	**-**	**-**	**8100**	**B003**

Modifying data in the database (UPDATE)

The UPDATE statement allows the contents of existing rows in a named table to be changed.

SYNTAX	**EXAMPLE**	
UPDATE TableName SET columnName1 = dataValue1 [, columnName2 = dataValue2 . . .] [WHERE searchCondition]	UPDATE Staff SET salary = 18,000, position='Manager' WHERE staffNo='SG44';	UPDATE Staff SET salary = salary*1.03 WHERE position=Assistant;

Output

staffNo	fName	lName	Position	Sex	DOB	Salary	branchNo
SG16	**Alan**	**Brown**	**Assistant**	**M**	**1957-05-25**	**8549**	**B003**
SG44	**Anne**	**Jones**	**Manager**	**-**	**-**	**18,000**	**B003**

Delete data from the database(DELETE)

The DELETE statement allows rows to be deleted from a named table.

SYNTAX	**EXAMPLE**
DELETE FROM TableName [WHERE searchCondition]	DELETE FROM Staff WHERE position = 'Manager';

Output

staffNo	fName	lName	Position	Sex	DOB	Salary	branchNo
SG16	Alan	Brown	Assistant	M	1957-05-25	8549	B003

To remove a relation from an SQL database

we use the drop table command. The drop table command deletes all information about the dropped relation from the database. The command

drop table r;

SQL is a non-procedural language, consisting of standard English words such as SELECT, INSERT, DELETE, that can be used by professionals and non-professionals alike. It is both the formal and de facto standard language for defining and manipulating relational databases.

Basic Structure of SQL Queries:

The basic structure of an SQL query consists of three clauses: select, from, and where. The query takes as its input the relations listed in the from clause, operates on them as specified in the where and select clauses, and then produces a relation as the result.

The sequence of processing in a SELECT statement is:

FROM specifies the table or tables to be used
WHERE filters the rows subject to some condition
GROUP BY forms groups of rows with the same column value
HAVING filters the groups subject to some condition
SELECT specifies which columns are to appear in the output
ORDER BY specifies the order of the output

The order of the clauses in the SELECT statement cannot be changed. The only two mandatory clauses are the first two: SELECT and FROM; the remainder are optional.

Retrieve all columns , all rows
Q1 :SELECT staffNo, fName, lName, position, sex, DOB, salary, branchNo FROM Staff;
Q2: SELECT * FROM Staff;
OUTPUT

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000.00	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000.00	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000.00	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000.00	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000.00	B005

Retrieve specific columns, all rows

Q1 : **SELECT** staffNo, fName, lName, salary **FROM** Staff;

OUTPUT

staffNo	fName	lName	salary
SL21	John	White	30000.00
SG37	Ann	Beech	12000.00
SG14	David	Ford	18000.00
SA9	Mary	Howe	9000.00
SG5	Susan	Brand	24000.00
SL41	Julie	Lee	9000.00

Use of DISTINCT

Retrieve specific columns, all rows

List the property numbers of all properties that have been viewed.

SELECT propertyNo FROM Viewing;	*SELECT DISTINCT propertyNo FROM Viewing;*

OUTPUT

propertyNo
PA14
PG4
PG4
PA14
PG36

propertyNo
PA14
PG4
PG36

Row selection (WHERE clause):

The above examples show the use of the SELECT statement to retrieve all rows from a table. However, we often need to restrict the rows that are retrieved. This can be achieved with the WHERE clause, which consists of the keyword WHERE followed by a search condition that specifies the rows to be retrieved.

Comparison of Search condition

List all staff with a salary greater than 10,000

SELECT staffNo, fName, lName, position, salary

FROM Staff WHERE salary > 10000;

OUTPUT

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00
SG37	Ann	Beech	Assistant	12000.00
SG14	David	Ford	Supervisor	18000.00
SG5	Susan	Brand	Manager	24000.00

List the addresses of all branch offices in London or Glasgow.

*SELECT * FROM Branch*

WHERE city = 'London' OR city = 'Glasgow';

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B003	163 Main St	Glasgow	G11 9QX
B002	56 Clover Dr	London	NW10 6EU

<table>
<tr><td>Range search condition (BETWEEN/NOT BETWEEN)</td><td>Set membership search condition (IN/NOT IN)</td></tr>
<tr><td>List all staff with a salary between 20,000 and 30,000.</td><td>List all managers and supervisors.</td></tr>
<tr><td>SELECT staffNo, fName, lName, position, salary FROM Staff WHERE salary BETWEEN 20000 AND 30000;</td><td>SELECT staffNo, fName, lName, position FROM Staff
WHERE position IN (‘Manager’, ‘Supervisor’);</td></tr>
<tr><td colspan="2">OUTPUT</td></tr>
<tr><td>
<table>
<tr><th>staffNo</th><th>fName</th><th>lName</th><th>position</th><th>salary</th></tr>
<tr><td>SL21
SG5</td><td>John
Susan</td><td>White
Brand</td><td>Manager
Manager</td><td>30000.00
24000.00</td></tr>
</table>
</td><td>
<table>
<tr><th>staffNo</th><th>fName</th><th>lName</th><th>position</th></tr>
<tr><td>SL21
SG14
SG5</td><td>John
David
Susan</td><td>White
Ford
Brand</td><td>Manager
Supervisor
Manager</td></tr>
</table>
</td></tr>
</table>

Using the SQL Aggregate Functions

The ISO standard defines five aggregate functions:

- COUNT – returns the number of values in a specified column;
- SUM – returns the sum of the values in a specified column;
- AVG – returns the average of the values in a specified column;
- MIN – returns the smallest value in a specified column;
- MAX – returns the largest value in a specified column.

These functions operate on a single column of a table and return a single value. COUNT, MIN, and MAX apply to both numeric and non-numeric fields, but SUM and AVG may be used on numeric fields only. Apart from COUNT(*), each function eliminates nulls first and operates only on the remaining non-null values.

Use of COUNT(*)	Use of COUNT(DISTINCT)
How many properties cost more than £350 per month to rent?	*How many different properties were viewed in May 2004?*
SELECT COUNT(*) AS myCount FROM PropertyForRent *WHERE rent > 350;*	*SELECT COUNT(DISTINCT propertyNo) AS myCount* *FROM Viewing WHERE viewDate BETWEEN '1-May-04' AND '31-May-04';*
OUTPUT	
myCount 5	myCount 2

Use of MIN, MAX, AVG

Find the minimum, maximum, and average staff salary.

SELECT MIN(salary) AS myMin, MAX(salary) AS myMax, AVG(salary) AS myAvg
FROM Staff;

myMin	myMax	myAvg
9000.00	30000.00	17000.00

Grouping Results (GROUP BY Clause)

A query that includes the GROUP BY clause is called a grouped query, because it groups the data from the SELECT table(s) and produces a single summary row for each group. The columns named in the GROUP BY clause are called the grouping columns. The ISO standard requires the SELECT clause and the GROUP BY clause to be closely integrated. When GROUP BY is used, each item in the SELECT list must be single-valued per group.

Use of GROUP BY
Find the number of staff working in each branch and the sum of their salaries.
SELECT branchNo, COUNT(staffNo) AS myCount, SUM(salary) AS mySum FROM Staff GROUP BY branchNo *ORDER BY branchNo;*
OUTPUT

branchNo	staffNo	salary
B003	SG37	12000.00
B003	SG14	18000.00
B003	SG5	24000.00
B005	SL21	30000.00
B005	SL41	9000.00
B007	SA9	9000.00

COUNT(staffNo)	SUM(salary)
3	54000.00
2	39000.00
1	9000.00

Using a subquery with equality

When a full SELECT statement is placed inside another SELECT statement, the outcome of this inner SELECT statement, also known as a subselect, is utilized in the outer statement to influence the composition of the ultimate result. These subselects, often referred to as subqueries or nested queries, can be employed in the WHERE and HAVING sections of an outer SELECT statement. Additionally, subselects may be found in INSERT, UPDATE, and DELETE statements.

List the staff who work in the branch at '163 Main St'.	
SELECT staffNo, fName, lName, position FROM Staff WHERE branchNo = (SELECT branchNo FROM Branch WHERE street = '163 Main St');	
OUTPUT	
The inner SELECT returns a result table containing a single value 'B003', corresponding to the branch at '163 Main St', and the outer SELECT becomes:	

SELECT staffNo, fName, lName, position

FROM Staff

WHERE branchNo = 'B003';

staffNo	fName	lName	position
SG37	Ann	Beech	Assistant
SG14	David	Ford	Supervisor
SG5	Susan	Brand	Manager

Multi-Table Queries

To combine columns from several tables into a result table we need to use a join operation. The SQL join operation combines information from two tables by forming pairs of related rows from the two tables. The row pairs that make up the joined table are those where the matching columns in each of the two tables have the same value.

List the names of all clients who have viewed a property along with any comment supplied.

SELECT c.clientNo, fName, lName, propertyNo, comment FROM Client c, Viewing v

WHERE c.clientNo = v.clientNo;

OUTPUT

clientNo	fName	lName	propertyNo	comment
CR56	Aline	Stewart	PG36	
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR62	Mary	Tregear	PA14	no dining room
CR76	John	Kay	PG4	too remote

For each branch, list the numbers and names of staff who manage properties, including the city in which the branch is located and the properties that the staff manage.

SELECT b.branchNo, b.city, s.staffNo, fName, lName, propertyNo FROM Branch b, Staff s, PropertyForRent p WHERE b.branchNo = s.branchNo AND s.staffNo = p.staffNo

ORDER BY b.branchNo, s.staffNo, propertyNo;

OUTPUT

The result table requires columns from three tables: Branch, Staff, and PropertyForRent, so a join must be used. The Branch and Staff details are joined using the condition (b.branchNo = s.branchNo),

to link each branch to the staff who work there. The Staff and PropertyForRent details are joined using the condition (s.staffNo = p.staffNo), to link staff to the properties they manage.

branchNo	city	staffNo	fName	lName	propertyNo
B003	Glasgow	SG14	David	Ford	PG16
B003	Glasgow	SG37	Ann	Beech	PG21
B003	Glasgow	SG37	Ann	Beech	PG36
B005	London	SL41	Julie	Lee	PL94
B007	Aberdeen	SA9	Mary	Howe	PA14

1.4.7 Database Normalization

Normalization theory is based on the concepts of normal forms. A relational table is said to be a particular normal form if it satisfied a certain set of constraints and the different level of normalization is shown Figure 1.20.

Functional Dependency is the basis for the first three normal forms.In a relational table, a column Y is considered functionally dependent on column X when the values in column Y can be uniquely identified by the values in column X.

Full functional dependence applies to tables with composite keys. In a relational table R, column Y is fully functionally dependent on X of R when it is functionally dependent on X and not functionally dependent on any subset of X.

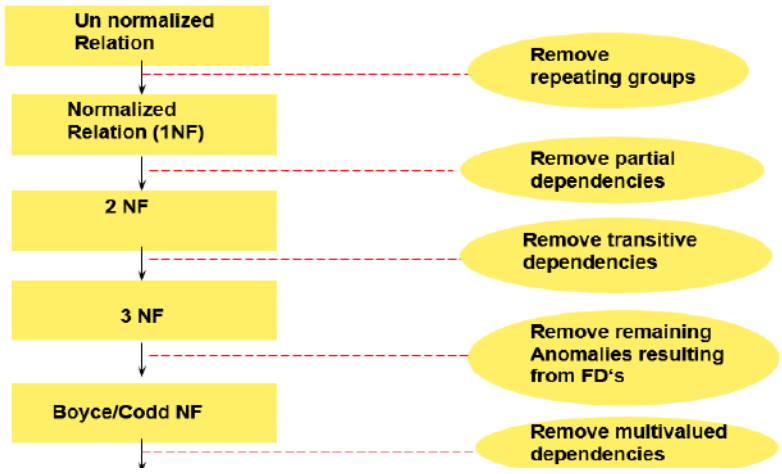


Figure 1.20. Different levels of Normalization

An Example: A company obtains parts from a number of suppliers. Each supplier is located in one city. A city can have more than one supplier located there and each city has a status code associated with it. Each supplier may provide many parts.

The company creates a simple relational table to store this information:

The company creates a simple relational table to store this information:

FIRST (s#, status, city, p#, qty)

s#	Supplier identification number
status	Status code assigned to city
City	City where supplier is located
p#	Part number of part supplied
Qty	Qty of parts supplied to date

Composite primary key is (s#, p#)

FIRST NORMAL FORM –1NF

A relational table is said to be in the first normal form if all values of the columns are atomic. That is, they contain no repeating values.

s#	city	status	p#	qty
s1	London	20	p1	300
s1	London	20	p2	100
s1	London	20	p3	200
s1	London	20	p4	100
s2	Paris	10	p1	250
s2	Paris	10	p3	100
s3	Tokyo	30	p2	300
s3	Tokyo	30	p4	200

SECOND NORMAL FORM – 2NF

Table FIRST contains redundant data. Redundancy causes update anomalies.

Update anomalies - problems that arise when information is inserted, deleted, or updated.

INSERT. The fact that a certain supplier (s5) is located in a particular city (Athens) cannotbe added until they supplied a part.

DELETE. If a row is deleted, then not only is the information about quantity and part lost but also information about the supplier.

UPDATE. If supplier s1 moved from London to New York, then six rows would have to be updated with this new information.

A relational table is in second normal form 2NF if it is in 1NF and every non-key column is fully dependent upon the primary key. That is, every non-key column must be dependent upon the entire primary key.

FIRST is in 1NF but not in 2NF because status and city are functionally dependent upon only on the column s# of the composite key (s#, p#).

Steps for transforming a 1NF table to 2NF is:

1. Identify any determinants other than the composite key, and the columns they determine.
2. Create and name a new table for each determinant and the unique columns it determines.
3. Move the determined columns from the original table to the new table. Determinate becomes the primary key of the new table.
4. Delete the columns you just moved from the original table except for the determinate which will serve as a foreign key.

PARTS

s#	p#	qty
s1	p1	300
s1	p2	100
s1	p3	200
s1	p4	100
s2	p1	250
s2	p3	100
s3	p2	300
s3	p4	200

SECOND

s#	city	status
s1	London	20
s2	Paris	10
s3	Tokyo	30

- **Modification Anomalies**
  - Tables in 2NF but not in 3NF still contain modification anomalies:
    - INSERT. The fact that a particular city has a certain status (Rome has a status of 50) cannot be inserted until there is a supplier in the city.
    - DELETE. Deleting any row in SUPPLIER destroys the status information about the city as well as the association between supplier and city.

THIRD NORMAL FORM – 3NF

A relational table is in third normal form (3NF) if it is already in 2NF and every non-key column is non transitively dependent upon its primary key. In other words, all non-key attributes are functionally dependent only upon the primary key.

SUPPLIER

s#	city	status
s1	London	20
s2	Paris	10
s3	Tokyo	30
s4	Paris	10

The table supplier is in 2NF but not in 3NF because it contains a ***transitive dependency***
SUPPLIER.s# —> SUPPLIER.city
SUPPLIER.city —> SUPPLIER.status
SUPPLIER.s# —> SUPPLIER.status

Steps for transforming a table into 3NF is:

- Identify any determinants, other the primary key, and the columns they determine.
- Create and name a new table for each determinant and the unique columns it determines.
- Move the determined columns from the original table to the new table. The determinan.t becomes the primary key of the new table.

The transformation of SUPPLIER into 3NF

SUPPLIER

s#	city
s1	London
s2	Paris
s3	Tokyo
s4	Paris
s5	London

CITY_STATUS

city	status
London	20
Paris	10
Tokyo	30
Rome	50

Advanced Forms: BOYCE CODD NORMAL FORM

Many practitioners argue that placing entities in 3NF is generally sufficient because it is rare that entities that are in 3NF are not also in 4NF and 5NF. The advanced forms of normalization are:

Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) is a more rigorous version of the 3NF. BCNF is based on the concept of determinants. A determinant column is one on which some of the columns are fully functionally dependent. A relational table is in BCNF if and only if every determinant is a candidate key.

BCNF is a more restrictive version of 3NF.

- The table must be in 3NF.
- Every non-trivial functional dependency must be a dependency on a superkey. (A trivial functional dependency would be A→A.)

Every relation in BCNF is also in 3NF, but the reverse is not necessarily true. 3NF allows attributes to be part of a candidate key that is not the primary key; BCNF does not.

This means that relations in 3NF are often in BCNF, but not always. The difference is subtle, but important, as in the next example.

Most relations that are in third normal form are also in Boyce-Codd normal form (BCNF), but not all. It might be easier to say what is not in BCNF: A relation is not in BCNF if

- the candidate keys in the relation are composite keys (that is, they are not single attribute keys)
- there is more than one candidate key in the relation
- the keys overlap, that is, some attributes in the keys are common.

BCNF Violation

The following is in 3NF but not in BCNF:

ID	TutorID	TutorSIN
8245	1254	000 737 999
9995	1567	000 656 999
5643	1254	000 737 999
6179	1742	000 651 999

Note that in the semantics for this data each student may have only one tutor, but each tutor may have many students.

This table is subject to insertion anomalies as both the Tutor ID and SIN must be entered whenever a tutor-student pair is entered. Here again, common sense tells us that this table is not well designed, even if we don't immediately recognize that it violates BCNF.

- Candidate keys are: {ID, TutorID} and {ID, TutorSIN}
- TutorID → TutorSIN and TutorSIN → TutorID, but because both TutorID and TutorSIN are prime attributes these FDs do not violate 3NF.
- Neither TutorID nor TutorSIN alone are superkeys, and thus BCNF is violated.

Both TutorID and TutorSIN are prime attributes because they are part of candidate keys. The use of both attributes in this table violates BCNF. The fix is to use only one of these two attributes. As noted before, BCNF is a stronger version of 3NF because it restricts both prime and non-prime attributes, while 3NF only restricts non-prime attributes.

BCNF Violation - Fixed

The original relation is decomposed into:

ID	TutorID
8245	1254
9995	1567
5643	1254
6179	1742

TutorID	TutorSIN
1254	000 737 999
1567	000 656 999
1742	000 651 999

Why this decomposition over the other, where student IDs are matched against tutor SINs? Here outside considerations help in our decision. SINs are treated specially and should be provided extra protection and privacy. The best way to do this is to minimize their use. Given a choice, then, between the two decompositions, the one that minimizes the use of SINs is the best.

Again, update and insertion anomalies have been removed.

Normalization Algorithm

1. Suppose that R is not in BCNF. Let X be a subset of R, A a single attribute in R, and X→A be a non-trivial FD that violates BCNF. Decompose R into R-A and XA.

If either R-A or XA are not in BCNF, repeat step 1 with the violating relation(s).

This simple recursive algorithm produces consistently BCNF compliant relations.Typically, databases development doesn't normally begin with a single, massive relation containing all the attributes.The organization of relations is guided by the semantics of the data. When semantics are unclear or a developer is simply unsure about the next steps, this algorithm can help the process.

Fourth Normal Form

The Fourth Normal Form (4NF) is a level of database normalization where there are no non-trivial multivalued dependencies other than a candidate key. It builds on the first three normal forms (1NF, 2NF, and 3NF) and the Boyce-Codd Normal Form (BCNF). It states that, in addition to a database meeting the requirements of BCNF, it must not contain more than one multivalued dependency.

Properties

A relation R is in 4NF if and only if the following conditions are satisfied:

1. It should be in the Boyce-Codd Normal Form (BCNF).
2. The table should not have any Multi-Valued Dependency.

A table with a multivalued dependency violates the normalization standard of the Fourth Normal Form (4NF) because it creates unnecessary redundancies and can contribute to inconsistent data. To bring this up to 4NF, it is necessary to break this information into two tables.

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU_ID, 21 contains two courses, Computer and Math and two hobbies, Dancing and Singing. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

So to make the table into 4NF, we can decompose it into two tables:

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Fifth Normal Form

The 5NF (Fifth Normal Form) is also known as project-join normal form. A relation is in Fifth Normal Form (5NF), if it is in 4NF, and won't have lossless decomposition into smaller tables. You can also consider that a relation is in 5NF, if the candidate key implies every join dependency in it.

Example

The below relation violates the Fifth Normal Form (5NF) of Normalization −

<Employee>

EmpName	EmpSkills	EmpJob (Assigned Work)
David	Java	E145
John	JavaScript	E146
Jamie	jQuery	E146
Emma	Java	E147

The relation can be decomposed into the following three tables; therefore, it is not in 5NF −

<EmployeeSkills>

EmpName	EmpSkills
David	Java
John	JavaScript
Jamie	jQuery
Emma	Java

The following is the <EmployeeJob> relation that displays the jobs assigned to each employee –

<EmployeeJob>

EmpName	EmpJob
David	E145
John	E146
Jamie	E146
Emma	E147

Here the skills that are related to the assigned jobs

<JobSkills>

EmpSkills	EmpJob
Java	E145
JavaScript	E146
jQuery	E146
Java	E147

Our Join Dependency –

{(EmpName, EmpSkills), (EmpName, EmpJob), (EmpSkills, EmpJob)}

The above relations have join dependency, so they are not in 5NF. That would mean that a join relation of the above three relations is equal to our original relation <Employee>.

1.5. Self-Assessment Question and Exercises

Self-Assessment Questions:

1. What does an entity represent in the Entity-Relationship Model?

   a. Database

   b. Table

   c. Object

   d. Relationship

2. In the Relational Data Model, what is a tuple?

   a. A set of attributes

   b. A row in a table

   c. A key constraint

   d. A unique identifier

3. In database normalization, which normal form ensures that there are no partial dependencies?

   a. First Normal Form (1NF)

   b. Second Normal Form (2NF)

   c. Third Normal Form (3NF)

   d. Boyce-Codd Normal Form (BCNF)

4. When representing a many-to-many relationship in a Relational Model, what is typically introduced?

   a. Foreign keys

   b. Composite keys

   c. Subqueries

   d. Indexes

5. What is a disadvantage of structured databases compared to unstructured databases?

   a. Lack of flexibility

   b. Difficulty in storing large amounts of data

   c. Limited support for complex relationships

   d. Inability to handle multimedia content

Short Questions:

1. Define a primary key in a relational database.
2. How are many-to-many relationships typically handled in a relational model?
3. Which SQL clause is used for filtering query results?
4. What is the purpose of normalization in database design?
5. Explain transitive dependency.
6. Name a disadvantage of structured databases compared to unstructured databases.
7. In data normalization, what does BCNF stand for?

Exercise:

1. Consider a university database with the following entities: Student, Course, and Instructor. Draw an Entity-Relationship Diagram (ERD) representing the relationships between these entities. Include attributes and cardinality.
2. Create a sample relational schema for the university database described in Exercise 1. Specify primary keys, foreign keys, and any necessary constraints.
3. Write an SQL query to retrieve the names of students who are enrolled in the course "Database Management." Assume relevant tables are present in the database.
4. Using Relational Algebra, express the following query: Retrieve the names of instructors who teach a course with more than 50 enrolled students.
5. Explain the difference between the UNION and JOIN operations in Relational Algebra. Provide a scenario where you would use the INTERSECT operation.
6. Define the term "data normalization" and explain its importance in database design.
7. Describe the difference between 2nd Normal Form (2NF) and 3rd Normal Form (3NF).

1.6. Summary

Thus, the Entity-Relationship Model (ER Model) and the Relational Data Model are foundational concepts in database management. The ER Model illustrates how entities relate to each other within a system using diagrams. On the other hand, the Relational Model represents and manages data in databases, focusing on tables. The mapping of ER models to the Relational Model is essential for creating structured databases that can be queried using specific languages. One fundamental query language is Relational Algebra, which aids in managing, retrieving, and performing operations on relational databases. Normalization is a crucial process in database design, used to avoid redundant data and update anomalies. It involves several stages or normal forms: First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF). Boyce-Codd Normal Form (BCNF) is a stricter version of 3NF, and Fifth Normal Form (5NF) addresses certain types of redundancy. These concepts, in unison, contribute to the creation of robust and efficient database systems.

Keywords

Entity-Relationship Model (ER Model), Relational Model, Query Languages, Relational Algebra, Normalization, 1NF, 2NF, 3NF, BCNF

Further readings

1. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan (2021), 7th edition, Database System Concepts.
2. Ramez Elmasri and Shamkant B. Navathe (2017), 7th edition, Fundamentals of Database Systems.
3. W3Schools SQL Tutorial, https://www.w3schools.com/sql/
4. https://www.geeksforgeeks.org/database-management-system-dbms/

UNIT 2

PARALLEL AND DISTRIBUTED DATABASES

CONTENTS

2.1 Introduction

Parallel and distributed databases are cornerstones in the world of big data and cloud computing. While parallel databases harness the power of multiple processors to execute operations concurrently for faster query responses, distributed databases store and manage data across different sites or networks, ensuring data availability and resilience. Both come with their unique challenges. For instance, parallel databases grapple with ensuring efficient I/O operations, while distributed databases face the hurdles of maintaining consistency and managing distributed transactions. Nevertheless, their combined strength and capabilities offer unmatched speed and scalability, catering to the demanding needs of modern data-intensive applications.

2.2 Learning Objectives

1. Understand the Fundamentals:
   - Differentiate between parallel and distributed databases.
   - Explain the advantages of using parallelism in database operations.
   - Describe the rationale behind the use of distributed databases.

2. Grasp the Mechanisms of Parallel Databases:
   - Detail the concept of I/O parallelism and its impact on performance.
   - Differentiate between Inter-Query and Intra-Query parallelism.
   - Understand the nuances of Inter-Operation and Intra-Operation parallelism.

3. Delve into Distributed Databases:
   - Explain the architecture of distributed data storage and its advantages.
   - Understand the complexities and methodologies of distributed transactions.
   - Describe the process and challenges of distributed query processing.

4. Appreciate Transaction Management:
   - Recognize the properties of distributed transaction management.
   - Explain the significance of concurrency control in ensuring data integrity.

5. Apply Knowledge in Real-world Scenarios:
   - Evaluate the benefits and challenges of implementing parallel or distributed databases in given business situations.
   - Suggest appropriate strategies for optimizing database operations based on the specific requirements and constraints of a scenario.

6. Engage in Continuous Learning:
   - Stay updated with the latest trends and advancements in the field of parallel and distributed databases.

- Participate in discussions, forums, and workshops to deepen understanding and share knowledge.

2.3 Overview Parallel and Distributed Databases

In today's data-driven era, organizations manage vast volumes of data, necessitating efficient storage and processing solutions. Parallel and distributed databases emerge as two principal approaches to address these demands, each offering distinct advantages and catering to specific scenarios.

Parallel Databases:

Definition: Parallel databases employ multiple processors and storage devices to execute database operations concurrently.

- **Key Features**:
  - I/O Parallelism: Enhances performance by enabling simultaneous disk operations.
  - Inter-Query & Intra-Query Parallelism: While the former executes multiple queries concurrently, the latter divides a single query for simultaneous execution.
  - Inter-Operation & Intra-Operation Parallelism: These deal with concurrent execution of different operations of varied queries and fragments of a single operation, respectively.
- **Benefits:** Improved query performance, efficient data loading, and heightened backup speed.

Distributed Databases:

Definition: A system wherein data storage devices are spread across a network, offering a unified view of the data regardless of its physical location.

- **Key Features:**
  - Distributed Data Storage: Data is scattered across different sites or physical locations, each equipped with its processing capabilities.
  - Distributed Transactions: Transactions spanning multiple sites, with the challenge of preserving ACID properties.

- Distributed Query Processing: Efficiently executing a query when data resides in various locations.

- **Benefits:** Enhanced data availability and reliability, modular development, and localized query processing reducing network traffic.

Challenges and Solutions:

- **Parallel Databases**: Ensuring even data distribution, managing hardware and software heterogeneity, and optimizing parallel query processing.
- **Distributed Databases:** Addressing data consistency across sites, managing distributed transactions, and safeguarding against potential site failures.

2.4 Parallel and Distributed Databases

2.4.1 Parallel Databases

In the past, parallel database systems were nearly disregarded, even by their most ardent supporters. However, today, they are prominently offered by almost every database system provider. This resurgence can be attributed to several factors:

- With increasing computer utilization, organizations' transaction needs have expanded. Additionally, the surge of the World Wide Web has led to websites with millions of users, generating vast databases due to the extensive data collection from these users.
- Organizations are now leveraging this vast amount of data, such as user purchase habits or web interactions, for strategic planning and pricing adjustments. These decision-support queries require processing immense data volumes, sometimes in the terabytes range, which single-processor systems cannot manage efficiently.
- The inherent structure of database queries is conducive to parallel processing. Both commercial and research-oriented systems have showcased the advantages and scalability offered by parallel query processing.
- The affordability of microprocessors has made parallel machines both commonplace and budget-friendly.

- Modern individual processors have evolved into parallel entities, thanks to multicore architectures.

In terms of parallel database system structures, there are three primary architectures:

Shared Memory Architecture

Shared memory architecture is a popular method in parallel processing where all processors have access to a unified memory pool, streamlining inter-processor communication and design. Among its key advantages are its inherent simplicity, as all processors share a single address space, which eases both design and programming. This shared space facilitates rapid data exchange between processors without the need to move data across a complex network. Moreover, this architecture supports uniform memory access times, simplifying algorithm designs. It is also scalable to an extent; adding more processors can be straightforward since they directly access the shared memory. Many modern systems also come with hardware-supported cache coherency, maintaining consistency across all processors.

However, shared memory architecture is not without its challenges. As the number of processors grows, memory access contention can arise, potentially slowing down the system. While it is scalable, there's a threshold beyond which adding more processors might lead to reduced performance due to this contention. Programmers need to grapple with the intricacies of synchronization mechanisms, like locks or semaphores, to maintain data consistency, adding layers of complexity. Additionally, a central memory fault can jeopardize the entire system, acting as a single point of failure. And from a financial perspective, the high-speed shared memory essential for efficient multi-processor access can be pricier than the local memory used in alternative architectures. The architecture of the Shared Memory is shown in Figure 2.1.

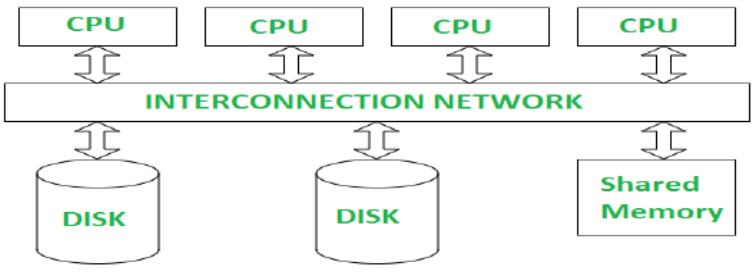


Figure 2.1 Shared Memory Architecture

Shared-disk architecture:

Shared-disk architecture is another prevalent paradigm in the realm of parallel processing, where multiple processors access and share a common disk storage, but each has its own private memory which is depicted in Figure 2.2. This model offers several notable advantages. Firstly, it provides inherent data redundancy; since all data resides on shared disks, any processor can access it, enhancing system resilience and availability. This architecture is inherently modular, allowing for easy system expansion by adding new processors without the need for data redistribution. Shared-disk systems also excel in workload balancing, as any processor can process any task as long as the data is accessible on the shared disks. Additionally, this setup simplifies data recovery and backup processes since the data storage is centralized.

However, the shared-disk model also brings its set of challenges. The most significant of these is the potential for disk contention, especially as the number of processors increases. With multiple processors trying to access the same disk resources, performance bottlenecks can emerge. Furthermore, ensuring data consistency becomes crucial, requiring sophisticated lock and synchronization mechanisms, which can add to system complexity. This also means that the architecture might need more advanced (and often more expensive) disk management and scheduling techniques to function optimally. Lastly, similar to shared-memory systems, the centralized nature of storage can introduce a single point of failure, although this can be mitigated with redundant storage solutions.

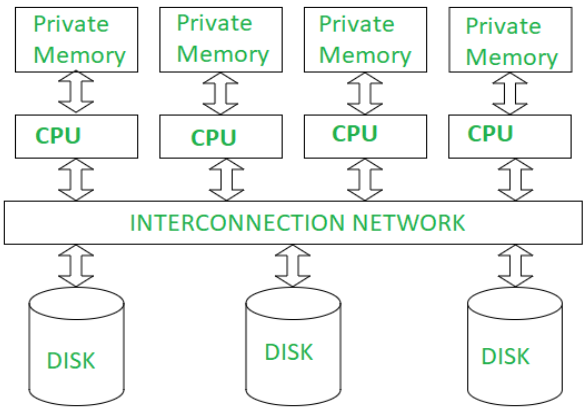


Figure 2.2 Shared-disk Architecture

Shared-nothing architecture

Shared-nothing architecture, a model where each node is self-sufficient with its own data storage and computing resources, stands out for its scalability and performance in large-scale database and processing tasks and the architecture diagram is given Figure 2.3. One of its paramount advantages is its excellent scalability; the architecture tends to scale linearly, as adding more nodes increases the system's storage and processing power proportionately without causing resource contention or significant overhead for coordination between nodes. This lack of shared resources virtually eliminates bottlenecks related to resource contention, a common issue in shared-disk or shared-memory setups. Furthermore, the decentralized nature of shared-nothing architecture enhances system reliability and availability; failure in one node doesn't cripple the entire system, and recovery processes are often faster and localized.

However, the architecture isn't without its downsides. Data distribution and repartitioning can be complex and resource-intensive, especially in scenarios where data needs to be rebalanced across nodes. This characteristic makes it less flexible for certain applications or unexpected loads. Additionally, the requirement for each node to be self-sufficient can lead to increased initial costs, as each must be equipped with its own set of hardware and software. Also, the system demands sophisticated software to manage data distribution, parallel computation, and fault tolerance across all nodes, which can introduce complexity in both development and operation phases. Despite these challenges, the shared-nothing architecture is widely recognized for its robust scalability and performance in handling massive data and high-demand applications.

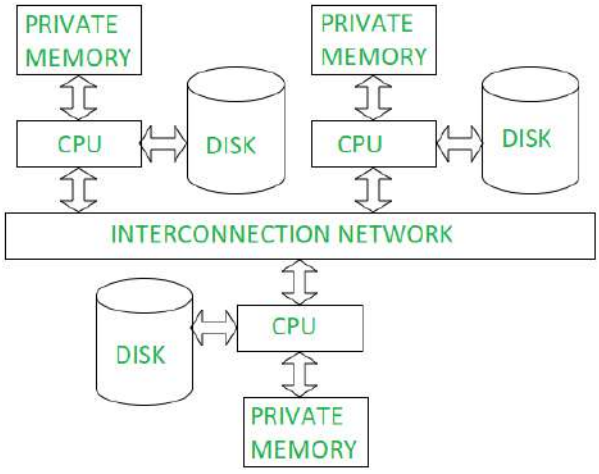


Figure 2.3 Shared-nothing Architecture

I/O Parallelism

I/O Parallelism refers to the concurrent execution of multiple input/output operations, allowing systems to read from and write to storage devices simultaneously, thereby significantly enhancing performance and throughput. It is particularly critical in database systems, where the speed of reading and writing data often determines the overall performance. In its simplest form, I/O parallelism refers to reducing the time required to retrieve relations from disk by partitioning the relations over multiple disks.

Horizontal Partitioning:

Horizontal partitioning involves dividing the rows of a table into smaller subsets, creating individual partitions containing distinct rows based on specified criteria. Each partition thus represents a horizontal slice of the original table. The key benefit is query efficiency; when searching for specific data subsets, the system can focus on relevant partitions, reducing the number of rows scanned. For instance, in a large database of global customers, horizontal partitioning could divide customers by country, ensuring that searches for customers in a particular nation only target the relevant partition.

Imagine an online retail database storing customer details. If this database serves customers worldwide, horizontal partitioning might be used to divide the customers based on continents (e.g., North America, Asia, Europe). Each partition would only contain customers from a particular continent. So, when querying for customers from Asia, only the Asia partition would be scanned, ensuring faster query times.

Vertical Partitioning:

In vertical partitioning, tables are segmented by columns rather than rows. Here, each partition holds a subset of the table's columns, effectively creating vertical slices of the original table. This method is particularly beneficial when different applications or queries require access to distinct sets of columns. For example, one application might only need customer names and email addresses, while another requires purchase histories. Vertical partitioning can isolate these columns, optimizing I/O for specific access patterns.

Consider a university database storing student information. Some operations might only need to access basic student information like name, ID, and date of birth, while other operations might

need detailed academic records. In this case, the database could be vertically partitioned into two: one partition containing basic student details and another containing academic details.

Range Partitioning:

Range partitioning segregates data based on defined value ranges within a specific column. Each partition is assigned a contiguous range of values, making this technique highly effective for range-based queries. Consider a financial database with transactions spanning multiple years. Range partitioning could distribute the data by year, so queries searching for transactions from a particular year only scan the corresponding partition, thus boosting query performance.

Consider a financial database recording transaction over several years. Using range partitioning, each year's transactions could be stored in a separate partition. For example, all transactions from 2020 would reside in one partition, while those from 2021 would be in another. When querying for all transactions in 2020, only the 2020 partition would be accessed.

Range partitioning is ideal for point and range queries on its attribute. Point queries use the partitioning vector to pinpoint the disk, while range queries determine potential disk ranges. This method often directs queries to fewer disks, boosting throughput and response time. However, large query ranges can cause I/O bottlenecks on limited disks, causing execution skew. In such scenarios, hash or round-robin partitioning, which utilize all disks, might offer swifter responses with similar throughput.

Hash Partitioning:

Hash partitioning determines data placement by employing a hash function on column values. This method aims to ensure a uniform distribution of data across partitions, especially when value ranges are unpredictable. For example, an e-commerce platform might use hash partitioning on user IDs to evenly distribute user data. With a balanced hash function, each partition would have roughly the same number of users, optimizing load distribution.

Hash partitioning excels in point queries on partitioning attributes and full-relation scans, ensuring even data distribution and efficient searches. However, it struggles with non-partitioning attribute queries and range queries due to its non-preservation of proximity.

Round-robin Partitioning:

Round-robin partitioning embodies simplicity. Data is evenly distributed across all partitions in a cyclical manner without adhering to specific criteria. This ensures a balanced distribution, making it an ideal choice for systems prioritizing load balancing in parallel processing scenarios. For example, in a parallel processing database, incoming data can be assigned to partitions in a round-robin fashion to evenly spread the load across storage resources.

Imagine a data streaming platform ingesting vast amounts of real-time data from various sensors. As data arrives, each new data point could be written to a partition in a round-robin fashion. For example, the first data point goes to Partition 1, the second to Partition 2, the third to Partition 3, and so on. Once the last partition is reached, the next data point cycles back to Partition 1.

Round-robin is perfect for applications requiring sequential full-relation reads per query, but complicates point and range queries, necessitating searches across all n disks. Partitioning type impacts relational operations like joins. The preferred partitioning method often depends on the operations executed. Typically, hash or range partitioning is favored over round-robin. For systems with multiple disks, the disk allocation depends on the relation size. Small relations fitting a single block are best on one disk while larger relations benefit from spanning all disks. If a relation occupies m blocks with n available disks, it should be distributed across the lesser of m or n disks.

Handling of Skew

In data partitioning, particularly methods beyond round-robin, there's a risk of skew—a disproportionate distribution of data across partitions. Skew can be:

1. **Attribute-value skew:** Where many tuples share a common partitioning attribute, leading to an uneven distribution.
2. **Partition skew**: An imbalance in data distribution across partitions, unrelated to specific attribute frequencies.

Regardless of the partitioning method (range or hash), attribute-value skew can emerge. With range partitioning, an ill-chosen partition vector can cause skew. However, with a well-defined hash function in hash partitioning, such skew is less likely. The repercussions of skew are

substantial, impacting system performance. As parallelism in a system increases, skew's detrimental effects become more pronounced. For instance, in a skewed distribution of 1000 tuples across ten sections, if one section holds disproportionally more tuples, the efficiency of parallel processing diminishes significantly.

To counter this, one approach is to employ sorting to create an even range-partitioning vector. Another is to utilize histograms, which provide insights into data distribution, making it easier to design balanced partitioning. Finally, the virtual processor method can be used. Here, tasks are allocated to virtual processors, which are then evenly mapped to actual processors, ensuring workload balance and offsetting the challenges of skew.

Inter-Query Parallelism

Interquery parallelism allows different queries or transactions to run concurrently. Although this increases transaction throughput, the response time for individual transactions remains unchanged compared to isolated execution. The primary objective of interquery parallelism is to enhance a system's capability to manage more transactions every second. This form of parallelism is more straightforward to implement in shared-memory parallel systems. Most database systems created for single-processor operations can adapt to this parallel architecture with minimal modifications. In a sequential setup where transactions operate concurrently in a time-shared manner, these transactions run parallel in a shared-memory architecture.

However, implementing interquery parallelism becomes challenging in shared-disk or shared-nothing architectures. Processors must collaboratively handle tasks like locking and logging, necessitating inter-processor communication. The system must ensure that data isn't updated simultaneously by two processors and that the latest data version is available in the processor's buffer pool. This challenge, known as the cache-coherency problem, requires specific protocols to maintain cache accuracy. Often, these protocols are merged with concurrency-control protocols to minimize overhead.

One such protocol for shared-disk systems is:

1. A transaction locks a page in shared or exclusive mode before any read or write access. After securing the lock, the transaction reads the latest page version from the shared disk.
2. Before releasing an exclusive lock on a page, the transaction saves the page to the shared disk and then releases the lock.

These actions ensure the transaction accesses the correct page version. Advanced protocols minimize disk read-write operations by retrieving the most recent page version from another processor's buffer pool, provided it's available. These protocols are tailored to manage simultaneous requests. In shared-nothing architectures, each page has an assigned home processor and is stored on a specific disk. Processors needing access to the page must request the home processor, as direct disk communication is impossible. Oracle and Oracle Rdb are examples of shared-disk parallel database systems that utilize interquery parallelism.

Interquery Parallelism Example: Online Shopping Database

Consider an online shopping database where customers are continually searching for products, adding items to their cart, and making purchases. Meanwhile, the system admin might be updating product prices, adding new stock, or removing outdated items. Additionally, the support team may be processing returns and refunds. All these actions involve multiple database queries.

- **Scenario without Interquery Parallelism:**

  If this online shopping database does not support interquery parallelism, then when Customer A is purchasing a laptop, Customer B might have to wait to search for headphones until Customer A's purchase is complete. Similarly, the admin updating the price of a smartphone might prevent Customer C from adding a tablet to their cart. In this scenario, transactions are processed one after another, leading to potential slowdowns during high traffic times, like Black Friday sales.

- **Scenario with Interquery Parallelism:**

  In a system with interquery parallelism, all the actions mentioned above can occur simultaneously. While Customer A's purchase is being processed, Customer B can still search for headphones, Customer C can add the tablet to their cart, and the system admin can update product prices. All these actions are treated as separate queries running in parallel. There's no need to wait for one transaction to complete before the next begins.

This parallel processing optimizes the system for high throughput, ensuring that the website can handle many users and actions at once. The response time for each individual transaction (like Customer A's purchase) might remain the same as if it were the only transaction occurring. The main advantage is the system's ability to accommodate and process multiple transactions simultaneously.

Intra-Query Parallelism

Intraquery parallelism involves executing a single query across multiple processors and storage devices, aiming to expedite time-consuming queries. Unlike interquery parallelism, where individual queries run one after another, intraquery parallelism focuses on concurrently processing parts of one query.

For example, if a query necessitates sorting a dataset, and this dataset is spread across multiple storage devices based on specific attributes, the sorting process can be expedited. Each section of the dataset can be sorted concurrently on its respective storage device. After sorting, the individual sections can be merged to produce the final sorted result.

There are two main strategies to achieve parallelism within a query:

- Intraoperation parallelism: We can speed up processing of a query by parallelizing the execution of each individual operation, such as sort, select, project, and join.
- Interoperation parallelism. We can speed up processing of a query by executing parallel the different operations in a query expression.

The two forms of parallelism are complementary, and can be used simultaneously on a query. Since the number of operations in a typical query is small, compared to the number of tuples processed by each operation, the first form of parallelism can scale better with increasing

parallelism. However, with the relatively small number of processors in typical parallel systems today, both forms of parallelism are important.

In the exploration of query parallelization, it's assumed that all queries are solely for reading purposes. The choice of parallelization algorithms hinges on the system's architecture. Instead of detailing algorithms for each architecture, focus is placed on the shared-nothing model. This means specifying when data needs to be transferred between processors. This model can be adapted to other architectures since data transfer can occur either through shared memory or shared disks, depending on the system. Thus, algorithms designed for the shared-nothing model can be applied elsewhere, and occasionally there are mentions of potential optimizations for different system types.

To streamline the explanations, assume there are 'n' processors, labeled P_0 to P_{n-1}, and an equal number of disks, D_0 to D_{n-1}. Every disk, D_i, is linked to a specific processor, P_i. Although many systems might have multiple disks for each processor, the algorithms can be adjusted to this setup by viewing D_i as a collection of disks. However, for this discussion, D_i is considered a single disk.

Intraoperation Parallelism

Relational operations can be parallelized by executing them on different subsets of large relations, offering a high degree of parallelism. This concept, known as intraoperation parallelism, is particularly effective in database systems.

Parallel Sort:

Consider a scenario where a relation needs to be sorted and is stored across n disks D_0 through D_{n-1}. If the relation is already range-partitioned based on the sorting attributes, each partition can be sorted independently, and the results then combined. This method capitalizes on parallel access, reducing the time needed to read the entire relation.

However, if the relation is partitioned differently, it can be sorted in two ways:

1. It can be range-partitioned based on the sorting attributes, followed by sorting each partition separately.
2. A parallel form of the external sort-merge algorithm can be employed.

For the range-partitioning sort, the process involves first range-partitioning the relation, then conducting individual sorts of each partition. This doesn't require using the same set of processors or disks initially storing the relation. Assuming processors P_0 through P_m are selected for sorting, and m is less than n, the operation involves two steps:

1. The tuples are redistributed according to a range-partition strategy, sending all tuples within a particular range to the corresponding processor, which temporarily saves them on its disk. This involves parallel reading of tuples and sending them to the designated processors, incurring disk I/O and communication overhead.
2. Each processor then sorts its own partition without needing to interact with others, an example of data parallelism. The final merge operation is simplified due to the initial range partitioning, ensuring the key values in one processor are always lower than those in any subsequent processor.

For balanced partitioning, a proficient range-partition vector is needed, ensuring each partition holds roughly the same number of tuples. Employing virtual processor partitioning can mitigate any imbalance.

An alternative method, **parallel external sort-merge**, can also be utilized regardless of how the relation has been initially partitioned. The process for this method is as follows:

1. Each processor sorts the data on its disk locally.
2. Sorted runs on each processor are then merged to produce the final sorted output.

The merge phase can also be parallelized:

1. Sorted partitions from each processor are range-partitioned across all processors, maintaining order so each processor receives sorted streams.
2. Every processor merges the incoming streams into a single sorted run.
3. The final result is obtained by concatenating the sorted runs on all processors.

However, this method can lead to execution skew. To counter this, each processor sends data blocks to each partition in rotation, ensuring parallel data reception. Certain machines, like those in the Teradata Purpose-Built Platform Family, use custom hardware for merging, where networks like BYNET merge outputs from various processors into a singular sorted output.

Parallel Join:

Parallel join algorithms distribute the task of evaluating tuple pairs against the join condition across multiple processors. If the tuples satisfy the join condition, they are included in the output.

Partitioned Join:

- Specifically, effective for equi-joins and natural joins.
- Two input relations, r and s, are partitioned into n partitions: r_0, r_1,..., r_{n-1} and s_0, s_1,..., s_{n-1}.
- Partitions r_i and s_i are sent to processor Pi for a local join computation.
- Partitioning is based on the same function on their join attributes, either range partitioning or hash partitioning.
- Post partitioning, any join technique (like hash join, merge join, nested-loop join) can be used locally.
- If the relations are pre-partitioned on the join attributes, work is reduced. Otherwise, they need to be repartitioned.
- Skew can be problematic in range partitioning. Hash partitioning might handle skew better unless many tuples have identical join attribute values.

The Figure 2.4 depicts the scenario behind the Partition and Parallel Join.

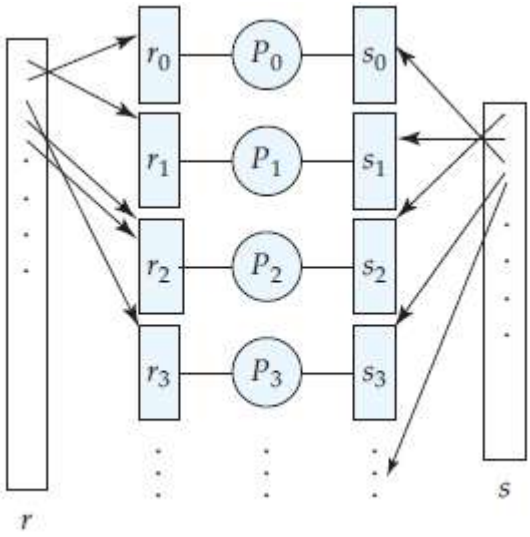
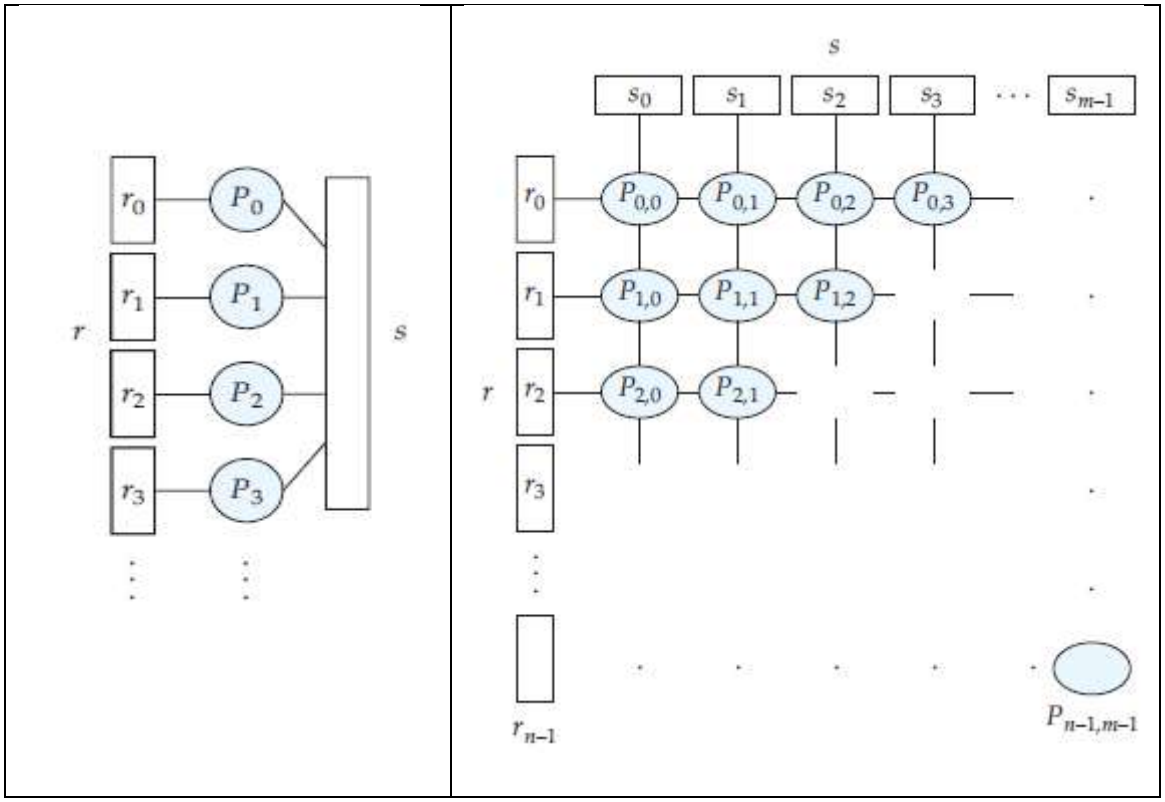


Figure 2.4 Partitioned Parallel Join.

Fragment and Replicate Join:

- Used when partitioning doesn't fit the join condition.
- In the asymmetric version:
  - Relation r is partitioned using any method.
  - Relation s is replicated across all processors.
  - Processor Pi computes the join of r_i with the entirety of s.
- In the general version:
  - Both relations, r and s, are partitioned into n and m partitions, respectively.
  - Every tuple in r can be tested with every tuple in s.
  - This method is effective for any join condition.

The Figure 2.5 depicts the scenario behind an Asymmetric fragment and Replicate. as well as Fragment and Replicate.



a. Asymmetric Fragment and Replicate	b. Fragment and Replicate
Figure 2.5	

Partitioned Parallel Hash Join:

1. Applied when parallelizing the hash join.
2. Smaller relation (say s) is chosen as the build relation.
3. A hash function (h1) maps each tuple in r and s to one of the n processors.
4. Each processor further partitions tuples using another hash function (h_2) for local hash join computation.
5. Both relations are distributed and repartitioned.
6. Each processor then executes the hash join locally.

Parallel Nested-Loop Join:

- Efficient when one relation (say s) is significantly smaller than the other (r).
- The small relation s is replicated while the existing partitioning of the larger relation r is used.
- Each processor executes an indexed nested-loop join of s with a specific partition of r.

In conclusion, parallel join algorithms aim to distribute the computational task across processors for efficiency. The choice of technique is influenced by the nature of the join and the size and characteristics of the relations being joined.

Cost of Parallel Evaluation of Operations:

In parallel processing, tasks are distributed among multiple processors to expedite the overall operation. Ideally, if there's perfect balance and no added costs, an operation utilizing ‘n’ processors would be ‘n’ times faster than using a single processor. When estimating the efficiency of such a setup, it's important to consider factors beyond just the basic operation's time cost.

Factors to be factored in include:

- **Start-up costs** for initiating the operation at multiple processors.
- **Skew** in the distribution of work among the processors, with some processors getting a larger number of tuples than others.
- **Contention for resources**—such as memory, disk, and the communication network—resulting in delays.
- **Cost of assembling** the final result by transmitting partial results from each processor.

The time taken by a parallel operation can be estimated as:

$$\mathbf{T_{part} + T_{asm} + max(T_0, T_1, \ldots, T_{n-1})}$$

Here, T_{part} is the relation partitioning duration, T_{asm} is the result compilation time, and T_i represents the duration for an operation on the i^{th} processor. If tuples are evenly distributed, each processor would theoretically manage 1/n of the total tuples.

However, this is a best-case scenario. Skew, or uneven distribution, is frequent and detrimental. Although dividing a query into parallel segments decreases the average segment size, the overall query processing time is constrained by the duration of the longest segment. Therefore, if work isn't evenly spread across processors, performance deteriorates. Skew during partitioning is analogous to the partition overflow problem in sequential hash joins. Strategies devised for hash joins, like overflow resolution and avoidance, can address skew arising from hash partitioning. Similarly, balanced range partitioning and virtual processor partitioning can counter skew induced by range partitioning.

Parallel processing in databases employs two main strategies: pipelined parallelism and independent parallelism.

Interoperations Parallelism

Pipelined Parallelism: Pipelining is a technique that helps optimize database queries. The basic concept is that while one operation, say A, is still generating its output, another operation, B, can start processing using the partial output from A. In sequential systems, the main benefit of this approach is that there's no need to save intermediate results on a disk. In parallel systems, pipelining is employed for much the same reason. Beyond this, it introduces a new dimension of

parallelism, drawing a parallel with instruction pipelines in hardware design. For instance, if we have operations A and B, they can run concurrently on separate processors. *B* consumes tuples in parallel with *A* producing them, his form of parallelism is called pipelined parallelism.

Take a multi-relational join: r1⋈ r2 ⋈ r3 ⋈ r4. The computations for these joins can happen in parallel across processors. If processor P1 handles temp1 ← r1 ⋈r2 and processor P2 takes care of r3 ⋈temp1, P1 can feed tuples to P2 even before it's done with its own computation. This ripple effect continues with P3 handling the subsequent join with r4.

While pipelined parallelism is beneficial with fewer processors, it has its limitations. Pipelines can't be too long, which limits parallelism. Some relational operations, like set-difference, can't be pipelined since they need all input before producing output. Furthermore, if one operation significantly outweighs others in execution cost, the speedup is minimal. In high parallelism scenarios, partitioning often takes precedence over pipelining. However, the primary advantage of pipelining remains: it prevents the need to write intermediate results to disk.

Independent Parallelism:

Operations within a query that don't rely on each other can run simultaneously, an approach known as independent parallelism. Taking our earlier join example: r1 ⋈r2 ⋈r3 ⋈r4, it's clear that temp1 ← r1 ⋈r2 can be calculated simultaneously with temp2 ← r3 ⋈r4. Once these are complete, we can compute their join: temp1 temp2. This process can further benefit from pipelining, feeding tuples from temp1 and temp2 into the final join operation.

Much like pipelined parallelism, independent parallelism is not the go-to solution in highly parallel systems due to its limited scalability. Nonetheless, it offers value in setups with a moderate degree of parallelism.

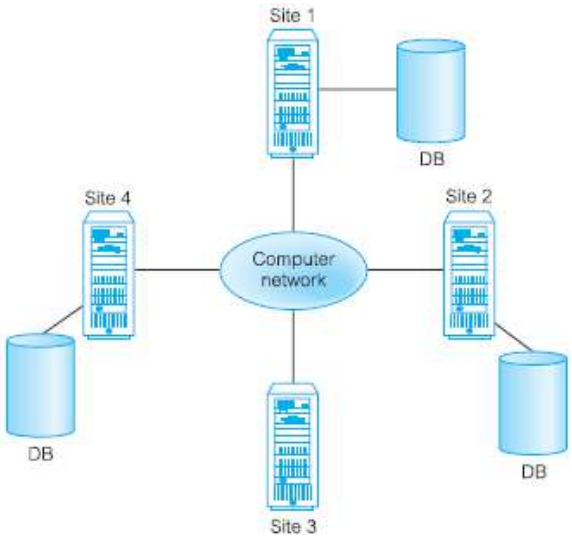
2.4.2 Distributed Database Architecture

A Distributed Database Management System (DDBMS) consists of a single logical database that is split into a number of fragments. Each fragment is stored on one or more computers under the control of a separate DBMS, with the computers connected by a communications network. Each site is capable of independently processing user requests that require access to local data (that is, each site has some degree of local autonomy) and is also capable of processing data stored on

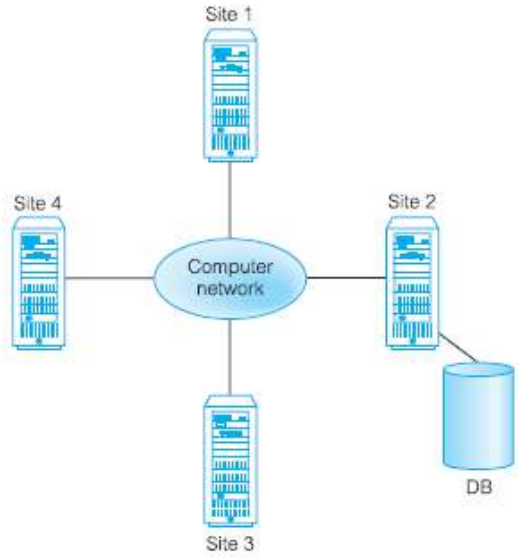
other computers in the network. Users access the distributed database via applications, which are classified as those that do not require data from other sites (local applications) and those that do require data from other sites (global applications). A DDBMS is required to have at least one global application.

A DDBMS therefore has the following characteristics:

- Collection of logically related shared data;
- The data is split into a number of fragments;
- Fragments may be replicated;
- Fragments/replicas are allocated to sites;
- The sites are linked by a communications network;
- The data at each site is under the control of a DBMS;
- The DBMS at each site can handle local applications, autonomously;
- Each DBMS participates in at least one global application.



a. Distributed database management system.



b. Distributed processing.

Figure 2.6

Distributed database: A logically interrelated collection of shared data (and a description of this data) physically distributed over a computer network.

Distributed DBMS: The software system that permits the management of the distributed database and makes the distribution transparent to users as shown in Figure 2.6.a.

Distributed processing: A centralized database that can be accessed over a computer network as shown in Figure 2.6.b.

Distributed DBMS and distributed processing are both paradigms that leverage multiple interconnected systems, but they target different challenges in the computing landscape. A Distributed DBMS focuses primarily on the storage, management, and access of data that's spread across various physical sites. Its goal is to present users with a unified database experience, even if the data resides in multiple locations. This helps in achieving redundancy, enhanced availability, and locality of data access. On the other hand, distributed processing emphasizes the distribution of computational tasks. Instead of centralizing data, it's about distributing the workload across multiple systems to achieve faster processing or task redundancy. In essence, while a Distributed DBMS is about where and how data is stored and accessed, distributed processing is about where and how tasks are executed.

Pros of Distributed Database:

1. **Availability**: If one node or site fails, the system can still function, ensuring high availability of data.
2. **Performance:** Data can be stored close to where it's frequently accessed, reducing query times and improving response rates.
3. **Scalability:** Distributed databases can scale out (horizontally) by adding more nodes, making it easier to handle increased loads.
4. **Redundancy:** Multiple copies of data can be maintained, which enhances data reliability.
5. **Modularity:** New sites or nodes can be easily integrated into the system without major changes.
6. **Local Autonomy:** Individual sites can maintain a level of independence, which can be useful in scenarios where local control over data is needed.

Cons of Distributed Database:

1. **Complexity:** Ensuring data consistency, managing transactions, and maintaining integrity across nodes can be challenging.
2. **Cost:** Setting up and maintaining a distributed system can be more expensive due to the requirement for additional hardware, network connections, and sophisticated management tools.
3. **Security Concerns:** Multiple sites mean multiple points of vulnerability, potentially increasing security risks.
4. **Network Dependence:** The system's functionality is heavily dependent on network reliability. Network failures can affect data access and consistency.
5. **Management Overhead:** Administering, monitoring, and managing backups across multiple sites can be cumbersome.
6. **Consistency Challenges:** Keeping data synchronized across all nodes, especially in systems where data is frequently updated, can be tricky.

Types of Distributed Database Management Systems

Distributed Database Management Systems (DDBMS) or DDMS can be categorized based on their architecture and the level of homogeneity. Here are the primary types of DDBMS:

1. **Homogeneous DDBMS:**

   All participating databases utilize the same underlying database management system (DBMS) software, adhere to the same data structures, and employ the same data models. Due to this uniformity across all nodes, they are easier to design and manage. Furthermore, transactions and operations are streamlined, as every node comprehends the same query language and structure and the architecture of Homogeneous Distributed DBMS is depicted in Figure 2.7.

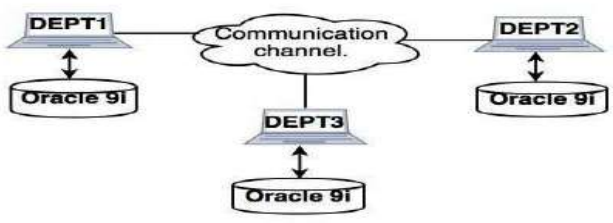


Figure 2.7 **Homogeneous Distributed DBMS**

2. Heterogeneous DDBMS:

The databases involved may have different DBMS software, data structures, and data models. For instance, one database might be running on Oracle while another is on MySQL. Requires additional middleware or software to translate between different database languages and structures. Integration and data translation become essential to ensure the seamless operation of the system. Figure 2.8 shows how the Heterogeneous Distributed DBMS is curated with SQL Server, MS Access and Oracle 9i.

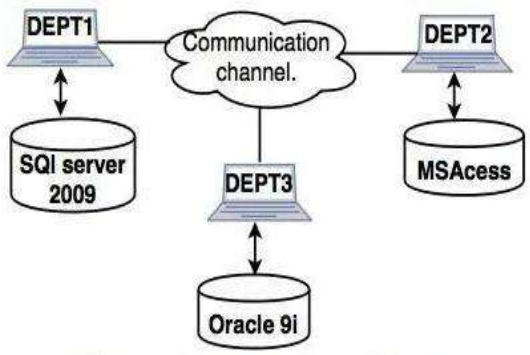


Figure 2.8 **Heterogeneous Distributed DBMS**

3. Federated (or Multidatabase) DDBMS:

A system that presents itself to users as one cohesive database, yet maintains some local independence. While it's a form of heterogeneous system, it grants greater autonomy to its constituent databases. This arrangement strikes a balance between self-rule and unification. While individual databases have their own degree of self-governance, the federated DBMS oversees global tasks and guarantees data consistency across the board.

4. Fully Distributed DDBMS:

This structure disperses data and DBMS software across various locations, yet to the users, it presents itself as a unified system. It boasts high resilience and availability, maintaining operations even when individual nodes encounter failures. The system frequently utilizes strategies like data replication and fragmentation to achieve its goals.

5. **Client-Server DDBMS:**

The system operates on a client-server paradigm, with clients initiating requests and servers handling and responding to these inquiries. This design permits a distinction between the application and the database, fostering better scalability and efficiency. The server's role to manage data processing and optimize queries.

6. **Peer-to-Peer DDBMS:**

Each point, or peer, within the system shares identical duties and roles. Absent of a central authority, each node has the capability to act as both client and server. The structure boasts impressive scalability and resilience. Its lack of a singular control center means it's less vulnerable to isolated failures.

Choosing the right type of DDBMS depends on the specific requirements, like the need for uniformity, flexibility, scalability, and the existing infrastructure.

Distributed Data Storage

Data can be stored across multiple sites using one of two primary methods:

1. **Replication**

Replication is a strategy wherein data is stored across multiple sites or nodes. Its primary goal is to enhance data reliability and availability. By distributing data across different locations, replication ensures that multiple users can access the data concurrently, thereby improving system performance. Essentially, the system maintains duplicate copies of data, ensuring that a comprehensive dataset is redundantly preserved at multiple sites. This enables users to retrieve consistent data from any location, ensuring uniformity and coherence.

However, replication isn't without its challenges. Frequent data updates are necessary to maintain consistency across all replicated sites. Moreover, the inherent nature of duplication means it requires significantly more storage space, which can be seen as a downside.

The common replication schemes are:

1. Full Replication: Data is entirely duplicated across all sites.
2. No Replication: Data is stored in a single location without any replication.
3. Partial Replication: Only a subset of the data is replicated across selected sites.

2. **Fragmentation:**

Here, datasets are divided into smaller parts, and each fragment is stored across different locations based on requirements. It's essential to ensure that these fragments, when combined, can reconstruct the original dataset without any loss of information. The advantage of fragmentation is that it eliminates data redundancy, hence issues related to data consistency are minimized.

1. **Usage**: Typically, applications interact with views rather than complete relations. As a result, distributing subsets of relations, instead of the entirety, aligns better with application needs and usage patterns.
2. **Efficiency**: Storing data near where it's most commonly accessed optimizes retrieval times and system responsiveness. Moreover, by not storing data that local applications don't require, we conserve storage resources.
3. **Parallelism**: Fragmentation allows transactions to be split into multiple subqueries that act on individual data fragments. This promotes concurrency in the system, enabling transactions that are safe to run in parallel to do so, enhancing overall system throughput.
4. **Security**: By not storing data that's unnecessary for local applications, it inherently becomes unavailable to potential unauthorized access, enhancing data security.
    1. However, fragmentation isn't without its challenges:
5. **Performance Concerns**: Global applications needing data from multiple fragmented locations might experience delays, reducing the speed of data retrieval and processing.
6. **Integrity Issues**: Ensuring data integrity can become more challenging when data and its associated dependencies are scattered across different sites. Keeping them consistent and avoiding anomalies requires additional effort and considerations.

Fragmentation in databases must be approached with precision. Three crucial rules ensure the correctness of fragmentation:

(1) **Completeness**: When a relation instance R is divided into fragments R_1, R_2,.. , R_n, every data item present in R should be included in at least one fragment. This rule guarantees that no data is lost during the fragmentation process.

(2) **Reconstruction**: There should be a definable relational operation that can reconstruct the relation R from its fragments. By doing so, this rule ensures the preservation of functional dependencies.

(3) **Disjointness**: A data item d_i that is part of fragment R_i should not be present in any other fragment. However, vertical fragmentation is an exception to this rule. In vertical fragmentation, primary key attributes need to be duplicated to facilitate reconstruction. This rule aims to ensure minimal data redundancy.

For horizontal fragmentation, a data item refers to a tuple. In the case of vertical fragmentation, a data item corresponds to an attribute.

Types of Fragmentation

Fragmentation in databases refers to dividing data into smaller, more manageable parts, based on certain criteria. Here are the primary types of fragmentation:

(1) **Horizontal Fragmentation**: In horizontal fragmentation, the data is divided based on subsets of tuples (or rows). Essentially, specific rows from a table are grouped and stored together based on some conditions or criteria. For instance, all customers from a particular region might be stored together in one fragment.

(2) **Vertical Fragmentation**: In this approach, data is fragmented based on subsets of attributes (or columns). This means specific columns from a table are grouped and stored together. For example, a table might be divided so that one fragment contains all customer names and addresses, while another contains their purchase histories. It's worth noting that in vertical fragmentation, the primary key is often duplicated across fragments to enable reconstruction.

(3) **Mixed Fragmentation**: This is a combination of both horizontal and vertical fragmentation. A table can first be horizontally fragmented based on a certain

criterion, and then each of those fragments might be further vertically fragmented based on different attributes.

The following Figure 2.9 depicts the idea behind the types of fragmentation and Figure 2.10 shows the fragmentation on a relation.

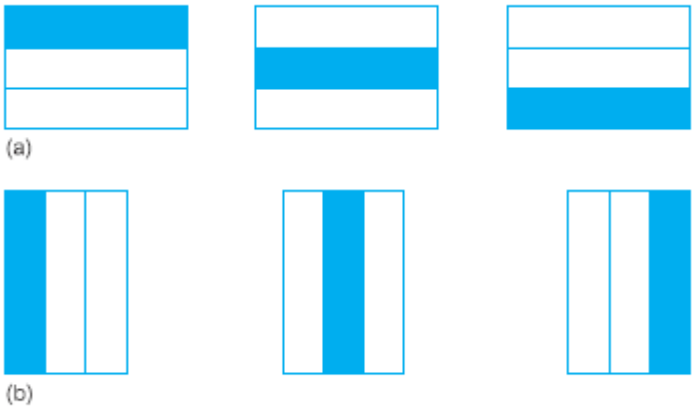


Fig 2.9 (a) Horizontal and (b) vertical fragmentation.

J

JNO	JNAME	BUDGET	LOC
J1	Instrumental	150,000	Montreal
J2	Database Dev.	135,000	New York
J3	CAD/CAM	250,000	New York
J4	Maintenance	350,000	Paris

Horizontal Partitioning

J1

JNO	JNAME	BUDGET	LOC
J1	Instrumental	150,000	Montreal
J2	Database Dev.	135,000	New York

J2

JNO	JNAME	BUDGET	LOC
J3	CAD/CAM	150,000	Montreal
J4	Maintenance.	310,000	Paris

Vertical Partitioning

JNO	BUDGET
J1	150,000
J2	135,000
J3	250,000
J4	310,000

JNO	JNAME	LOC
J1	Instrumentation	Montreal
J2	Database Devl	New York
J3	CAD/CAM	New York
J4	Maintenance	Paris

Figure 2.10 Example of Horizontal and Vertical Partitioning

Mixed fragmentation is the result of applying both horizontal and vertical fragmentation. The concept of mixed fragmentation is depicted in Figure 2.11, and an example of the same on a relation is shown in Figure 2.12.

Figure 2.11 Mixed fragmentation: (a) vertical fragments, horizontally fragmented;

(b) horizontal fragments, vertically fragmented.

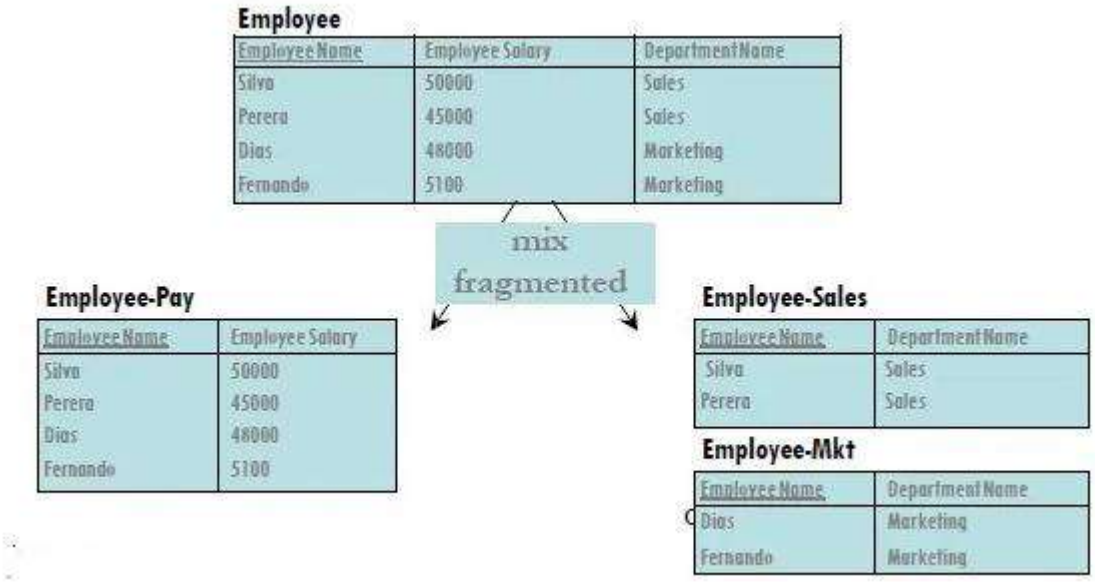


Figure 2.12 Mixed Fragmentation – Example

2.4.3 Distributed Transactions

A distributed transaction is a transaction that involves multiple data sources, possibly spread across different locations in a distributed database system or even across multiple distinct systems. It encompasses operations that, despite their distribution, need to be executed in a coordinated and atomic manner. The primary goal is to ensure that even when parts of the transaction are carried out on different servers or databases, the entire transaction should either complete successfully (commit) or fail entirely (rollback), preserving the principle of atomicity.

key aspects of distributed transactions:

1. **Atomicity**: As with all transactions, atomicity ensures that all operations within the distributed transaction either complete successfully or none of them do. In other words, if any part of the distributed transaction fails, the entire transaction is rolled back.
2. **Two-phase Commit (2PC)**: This is a standard protocol used to achieve atomicity in distributed transactions. The protocol involves two phases - a preparation phase and a commit phase. All involved parties (or nodes) agree to commit or abort the transaction based on these phases.
3. **Distributed Deadlocks**: In distributed transactions, deadlocks (situations where transactions are unable to proceed because of cyclic dependencies) can be more challenging to detect and resolve due to the distribution of operations across different nodes.
4. **Concurrency Control**: Ensuring the isolated execution of distributed transactions is crucial. Various techniques and protocols, like distributed timestamp ordering or distributed two-phase locking, can be employed to achieve this.
5. **Recovery**: In the event of system failures, mechanisms must be in place to recover the data in a consistent state. The recovery process in distributed transactions can be more complex due to the involvement of multiple nodes.
6. **Transparency**: From the perspective of the end-user or application, distributed transactions should appear as regular, local transactions. This abstraction hides the complexities of the underlying distributed system.

Distributed transactions are fundamental in environments where data is spread across multiple locations or systems but needs to be accessed and manipulated in a unified, consistent manner. Examples include distributed databases, banking systems, and modern microservices architectures.

In centralized DBMS, four primary modules were identified: the transaction manager, scheduler, recovery manager, and buffer manager. The transaction manager coordinates transactions based on application requirements, working in tandem with the scheduler, which aims to optimize concurrency without compromising database consistency. If a transaction

failure occurs, the recovery manager steps in to restore the database to its pre-transaction state, ensuring consistency. Post any system failure, the recovery manager also ensures database stability. Additionally, the buffer manager oversees the efficient data movement between disk storage and main memory.

Access to data items in a distributed system is typically facilitated through transactions that must uphold the ACID properties. Two primary types of transactions exist: **local and globa**l. Local transactions pertain to and modify data in a singular local database, whereas global transactions span and operate across multiple local databases. The complexity intensifies when multiple sites participate in execution. Failures at any site, or disruptions in the communication links between them, have the potential to result in erroneous computations.

In a distributed DBMS context, each local DBMS contains these modules. Additionally, every site has a global transaction manager or a transaction coordinator to oversee both global and local transaction executions. Instead of direct communication between transaction managers at different sites, the data communications component facilitates inter-site communication.

To execute a global transaction originating from site S1, the following steps are followed:

1. The transaction coordinator at site S1 (TC1) segments the transaction into several subtransactions, utilizing information from the global system catalog.
2. Using the data communications component at site S1, the subtransactions are dispatched to the designated sites, for instance, S2 and S3.
3. The transaction coordinators at sites S2 and S3 oversee these subtransactions. The outcomes of the subtransactions are relayed back to TC_1 through the data communications components.

This procedure is illustrated in Figure 2.13.

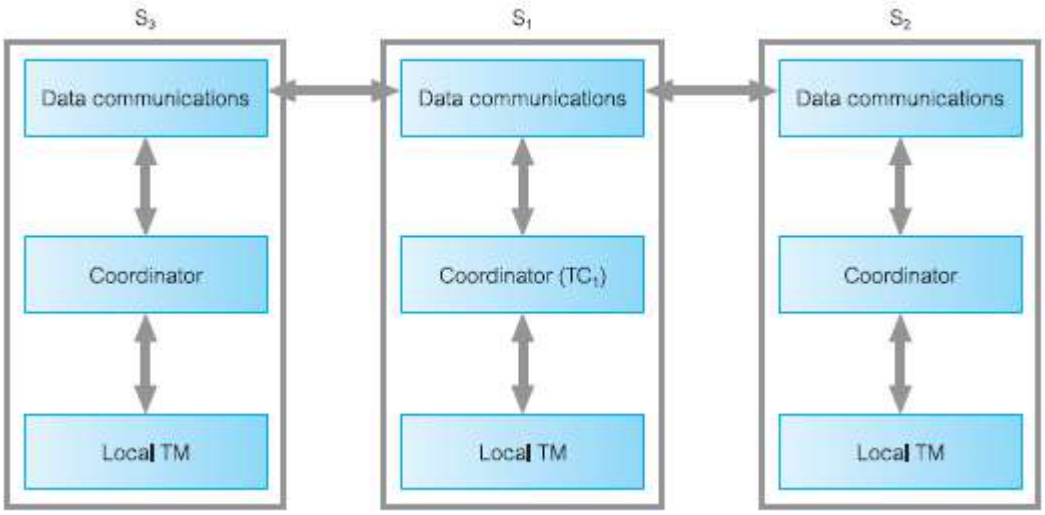


Figure 2.13 Coordination of distributed transaction.

System Failure Modes

Distributed systems are susceptible to the same types of failures as centralized systems, such as software and hardware errors or disk crashes. However, distributed environments introduce additional unique failures. The primary failure types include:

1. Site failure.
2. Message loss.
3. Communication link failure.
4. Network partitioning.

Message loss or corruption is a potential issue in distributed systems. Transmission-control protocols, like TCP/IP, are employed to address these errors. When sites A and B lack a direct connection, messages between them need to traverse through a series of communication links. A failure in any of these links requires the rerouting of messages. While alternative routes might exist in certain situations, allowing for uninterrupted communication, other scenarios might result in some site pairs losing their connection.

2.4.4 Distributed query Processing

Distributed query processing refers to the mechanisms and strategies used to execute a database query across multiple interconnected databases or nodes in a distributed database system. The goal is to efficiently retrieve data from multiple locations and present it as a single unified result. Distributed query processing poses several challenges due to the distribution of data, heterogeneity of systems, and issues of network latency and failure.

- Goals of Distributed Query Processing:
  - Efficiency: Minimize the total cost (computation, I/O, and communication).
  - Optimality: Find the best execution strategy from all possible plans.
  - Transparency: Users shouldn't need to be aware of where data resides.
- Challenges:
  - Data Location: Determining where the required data resides.
  - Network Costs: Sending and receiving data over a network often incurs more cost than local processing.
  - Data Replication: Handling data that exists in multiple locations.
  - Concurrency and Reliability: Addressing potential network failures and ensuring concurrent access doesn’t compromise data integrity.
- Steps in Distributed Query Processing:
  - Query Decomposition: Break the high-level query (e.g., SQL) into a series of low-level operations (Relational Algebra).
  - Data Localization: Identify where data resides and which fragments or replicas are relevant.
  - Global Optimization: Generate possible strategies or query plans and evaluate them based on a cost model. This includes decisions like which operations to perform locally and which data to transfer.
  - Local Optimization: Once a global plan is set, further optimizations are done at each local site.
- Optimization Techniques:
  - Join Strategies: Depending on data location and size, different join techniques like semi-join, broadcast join, or parallel join can be used.

- Pushing Selections: Apply filtering conditions (selections) as early as possible, preferably before data transfer.
- Reduction of Data Movement: Minimize data movement across the network by processing data locally whenever possible.
- Parallel Execution: Leverage parallelism inherent in distributed systems to execute multiple operations simultaneously.

- Tools & Systems: There are various database systems that provide support for distributed query processing, such as:
  - RDBMSs like Oracle, PostgreSQL, and Microsoft SQL Server with extensions for distributed processing.
  - NoSQL Databases like Cassandra, MongoDB, and Couchbase that inherently support distributed processing.
  - NewSQL Databases like CockroachDB and Google Spanner designed for distributed and scalable processing.

For centralized systems, the primary metric for evaluating the cost of a specific strategy hinges on the number of disk accesses. However, in a distributed system, other factors come into play, such as:

- The expense associated with transmitting data across the network.
- The potential performance boost from enabling multiple sites to process segments of the query simultaneously.

The cost disparity between data transfer over the network and from the disk can vary considerably based on the network's nature and the disks' speed. Therefore, it's not feasible to focus exclusively on either disk or network expenses. Instead, a balanced consideration of both is essential.

Query Transformation

Consider a simple query: '**Find all the tuples in the account relation**'. Despite the simplicity of the query, processing it is not trivial, since the account relation might be fragmented, replicated, or both. If the account relation is replicated, we must choose from among the replicas. If none of the replicas are fragmented, we select the one with the lowest transmission cost. However, if a replica is fragmented, the choice becomes more complicated, as we need to compute several joins or unions to reconstruct the account relation. In this scenario, the number of strategies for our

straightforward example could be substantial. Exhaustively enumerating all alternative strategies for query optimization may not be practical in such situations.

Fragmentation transparency implies that a user can write a query such as

$$\sigma_{branch\_name = \text{“Hillside”}} (account)$$

Since *account* is defined as:

$$account_1 \cup account_2$$

The expression that results from the name translation scheme is:

$$\sigma_{branch\_name = \text{“Hillside”}} (account_1 \cup account_2)$$

Using the query-optimization techniques, we can simplify the preceding expression automatically. The result is the expression:

$$\sigma_{branch\_name = \text{“Hillside”}} (account_1) \cup \sigma_{branch\_name = \text{“Hillside”}} (account_2)$$

which includes two subexpressions. The first involves only $account_1$, and thus can be evaluated at the Hillside site. The second involves only $account_2$, and thus can be evaluated at the Valleyview site. There is a further optimization that can be made in evaluating:

$$\sigma_{branch\_name = \text{“Hillside”}} (account_1)$$

Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation. In evaluating:

$$\sigma_{branch\_name = \text{“Hillside”}} (account_2)$$

we can apply the definition of the $account_2$ fragment to obtain:

$$\sigma_{branch\_name = \text{“Hillside”}} (\sigma_{branch\_name = \text{“Valleyview”}} (account))$$

This expression is the empty set, regardless of the contents of the *account* relation. Thus, our final strategy is for the Hillside site to return *account*1 as the result of the query.

Simple Join Processing

A major decision in the selection of a query-processing strategy is choosing a join strategy. Consider the following relational-algebra expression**:**

$$account \bowtie depositor \bowtie branch$$

Assume that the three relations are neither replicated nor fragmented, and that *account* is stored at site S_1, *depositor* at S_2, and *branch* at S_3. Let S_I denote the site at which the query was issued. The system needs to produce the result at site S_I. Among the possible strategies for processing this query are these:

- Ship copies of all three relations to site S_I. choose a strategy for processing the entire query locally at site S_I .
- Ship a copy of the *account* relation to site S_2, and compute $temp_1 = account \bowtie depositor$ at S_2. Ship *temp*1 from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie branch$ at S_3. Ship the result $temp_2$ to S_I .
- Devise strategies similar to the previous one, with the roles of S_1, S_2, S_3 exchanged.

No single strategy is consistently the best. Factors to consider include the volume of data being transferred, the cost of transmitting a data block between two sites, and the processing speed at each site. Consider the first two strategies listed. If indices at S_2 and S_3 are beneficial for computing the join, and all three relations are transferred to S_I, either these indices need to be recreated at S_I or a different, potentially costlier, join strategy must be adopted. Recreating indices introduces additional processing overhead and disk accesses. In the second strategy, a potentially large relation (account-depositor) has to be transferred from S_2 to S_3. This relation repeats a customer's name for each account they possess. Therefore, the second strategy might result in more network transmission compared to the first.

Semi-join Strategy

Assume there's a need to assess the expression r1 combined with r2, with r_1 located at site S_1 and r_2 at site S_2. The corresponding schemas for r_1 and r_2 are R_1 and R_2. The goal is to obtain the final result at S_1. If numerous tuples of r_2 don't combine with r_1's tuples, then transferring r_2 to S_1 would

mean transporting tuples that don't add value to the outcome. The aim is to eliminate such tuples prior to moving data to S_1, especially when network transmission expenses are considerable.

A suggested strategy for this could be:

1. Compute $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$ at S_1.
2. Ship $temp_1$ from S_1 to S_2.
3. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ at S_2.
4. Ship $temp_2$ from S_2 to S_1.
5. Compute $r_1 \bowtie temp_2$ at S_1. The resulting relation is the same as $r_1 \bowtie r_2$.

To evaluate this strategy's effectiveness, ensure it produces the desired outcome. Using this strategy, especially when a limited number of r_2 tuples join, can be beneficial. This is probable when r_1 stems from a relational-algebra expression involving a selection. Hence, $temp_2$ might contain way fewer tuples than r_2, leading to cost savings due to the reduced data transfer.

This approach is termed the semi-join method, inspired by the relational algebra's semi-join operation. Essentially, the semi-join of r_1 with r_2 extracts tuples from r_1 that participate in the combined result of r_1 and r_2. This method can be expanded for multiple relation joins. There's a significant amount of theory on using semi-joins for optimizing queries

Join Strategies that Exploit Parallelism

Consider a scenario where four relations need to be joined: r_1 combined with r_2, further combined with r_3, and then with r_4. Each relation r_i is located at its respective site S_i. The desired outcome is for the final result to be displayed at site S1. Several strategies can be employed for simultaneous evaluation.

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

In one approach, r_1 is transferred to S_2, and the combination of r_1 and r_2 is computed there. Concurrently, r_3 is moved to S_4, where r_3 and r_4 are computed together. Instead of waiting for the entire join process to finish at S_2, it can immediately send tuples of the combined r_1 and r_2 to S_1. Similarly, S_4 can send tuples of the combined r_3 and r_4 to S_1. As soon as the tuples from both these combined relations reach S_1, the calculation of their joint result begins. This means that the final join result's computation at S1 can occur simultaneously with the computations at S_2 and S_4.

Distributed Transaction Management

A distributed database system is composed of multiple database sites, each featuring a centralized database system. These sites are interconnected through a communication network, allowing them to exchange messages. Within this framework, a transaction is defined as a series of actions conducted on data objects. Conflicting operations occur when two tasks, oi(x) and oj(x), access the same data object x, and at least one of them involves a write operation.

To manage both local and global transactions effectively, each site within a distributed database system is equipped with three key components: a transaction manager, a transaction coordinator, and a recovery manager. The transaction manager oversees the execution of transactions at its respective site, while also keeping a log to aid in potential recovery processes. It employs concurrency control methods to ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties are upheld during the simultaneous execution of multiple transactions.

Meanwhile, the transaction coordinator plays a strategic role, organizing and directing sub-transactions that are carried out across various sites. It's also tasked with deciding whether multi-site transactions should be committed or aborted. On the other hand, the recovery manager serves as a safety net, prepared to restore the database to a stable state in the event of a failure.

This system's architecture, illustrated in Figure 2.14, highlights the distinct components involved in transaction management at every database site. Distributed transaction management is crucial to the coherence and reliability of distributed database systems, adapting various concurrency control strategies, commit protocols, and recovery methods to suit the unique requirements of different distributed database configurations.

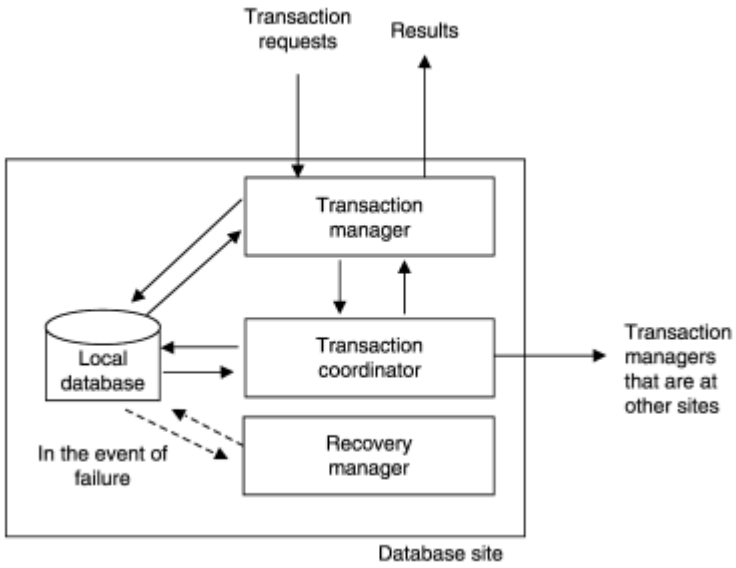


Figure 2.14 Distinct Components of Transaction Management

ACID Properties

A DBMS manages data to ensure its integrity, especially during modifications. If data integrity is compromised, the entire dataset can become corrupted. To maintain data integrity in a database management system, four essential properties, known as the ACID properties, come into play. These properties are particularly significant for transactions that undergo various tasks. This section discusses the examples of each property, and Figure 2.15 shows the four ACID properties.

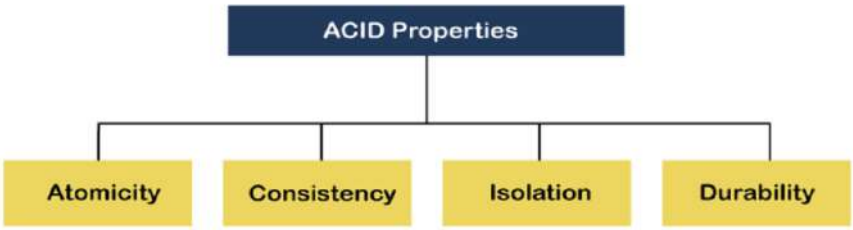


Figure 2.15 ACID Properties

ATOMICITY: Atomicity ensures that data operations are atomic, meaning they are indivisible and irreducible. In other words, if any operation is performed on data, it should either be executed completely or not executed at all. There should be no scenario where an operation is partially executed. This principle is particularly relevant for transactional operations.

Example: Suppose Remo has an account (Account A) with a balance of $30. He wants to transfer $10 to Sheero's account (Account B), which currently holds $100. If the transfer is successful, Account A should be debited by $10, reducing its balance to $20, and Account B should be credited by $10, increasing its balance to $110. Now, consider a situation where the debit operation from Account A is successful, but the credit operation to Account B fails for some reason. As a result, $10 is subtracted from Remo's account, reducing it to $20, but Sheero's account remains unchanged at $100. This scenario illustrates the importance of atomicity where the entire transaction should be either complete successfully or not execute at all, as depicted in figure 2.16.

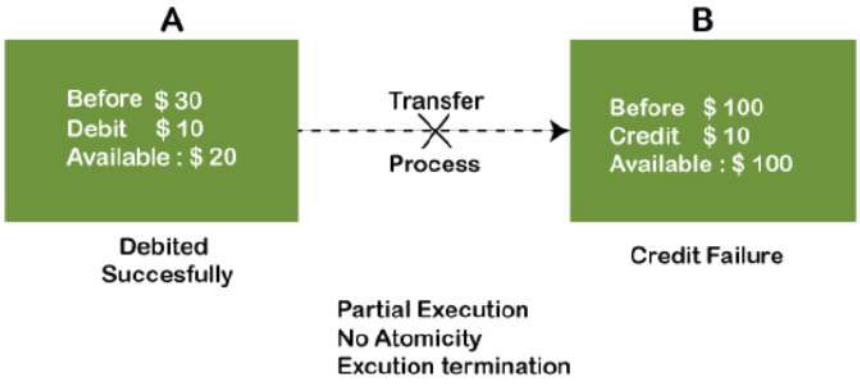


Figure 2.16 Atomicity on a Transaction (Failure)

In the diagram 2.16, the balance in account B remains unchanged at $100 even after attempting to credit $10. This indicates that the transaction is not atomic. In contrast, the subsequent Figure 2.17, demonstrates that both the debit and credit operations have been executed successfully, confirming that the transaction adheres to atomicity.

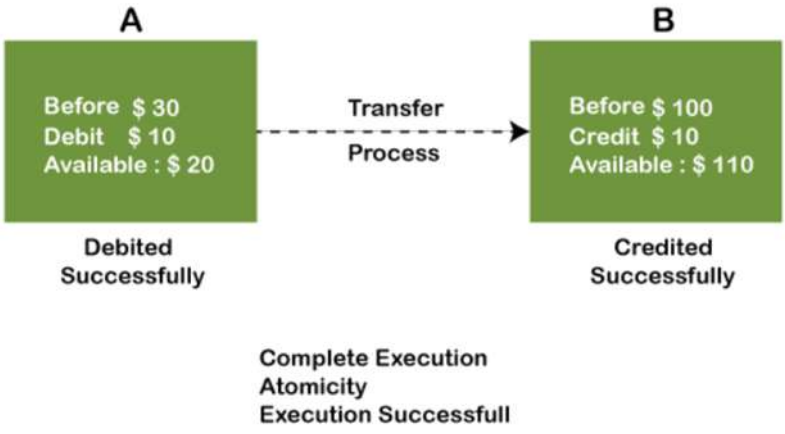


Figure 2.17 Atomicity on a Transaction (Success)

When atomicity is compromised in banking systems, it can lead to significant issues. Therefore, ensuring atomicity is a primary focus in these systems.

CONSISTENCY:

Consistency implies that values should always be preserved. In a DBMS, it is essential to maintain data integrity. This means if any change is made to the database, it should always be preserved. Especially in the context of transactions, data integrity is crucial to ensure the database remains consistent both before and after the transaction. The data must always be accurate. The sample transaction which ensures consistency is shown in Figure 2.18.

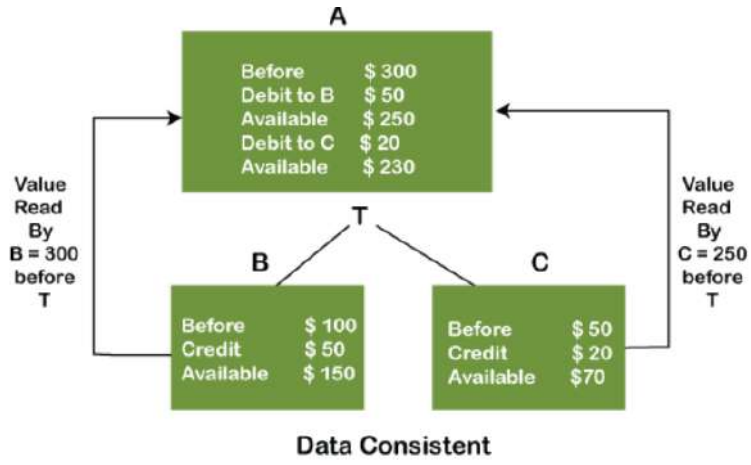


Figure 2.18 Consistency on a Transaction

In the figure 2.18, there are three accounts: A, B, and C. Account A initiates a transaction T to both B and C sequentially. Two operations take place: Debit from account A and Credit to either account B or C. Initially, account A debits $50 to account B. Before the transaction, account B reads the balance of account A as $300. After the successful completion of transaction T, the balance in account B increases to $150. Next, account A debits $20 to account C. At this time, account C reads account A's balance as $250, which is correct since $50 has already been successfully debited to account B. The debit and credit operations between account A and C proceed without issue. As we can see, the transaction is executed successfully, and the balances are read correctly, ensuring data consistency. If both accounts B and C were to read account A's balance as $300, it would imply data inconsistency because the subsequent debit operations would result in an inconsistency.

ISOLATION:

The term 'isolation' denotes separation. In DBMS, isolation ensures that concurrent operations on different datasets don't interfere with each other. Simply put, an operation on one dataset should only commence once operations on another dataset are complete. This means if two operations are carried out on two different datasets, they shouldn't influence each other's values. When multiple transactions occur simultaneously, it's vital to maintain data consistency. Changes made within a particular transaction shouldn't be visible to other transactions until they are committed to memory.

Example: Consider two operations running concurrently on separate accounts. The values in these accounts should remain unaffected by each other, ensuring persistence. As illustrated in the diagram below, account A initiates two transactions, T1 and T2, to accounts B and C respectively. Both transactions execute independently, without influencing one another. This characteristic exemplifies isolation and the Figure 2.19 shows the Isolation property.

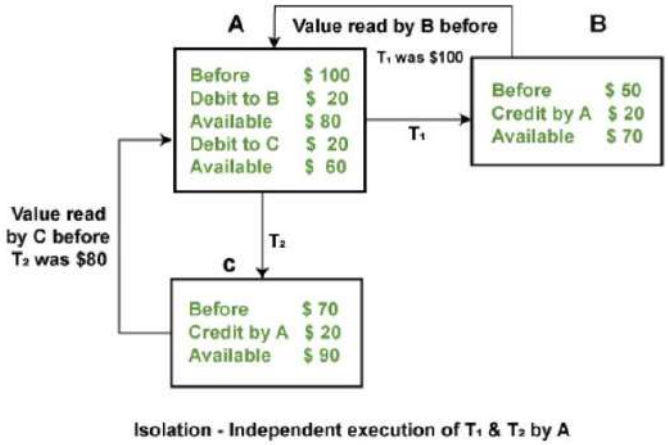


Figure 2.19 Isolation on a Transaction

DURABILITY:

Durability guarantees persistence. In the context of DBMS, durability ensures that once an operation is successfully executed, its effects become permanent in the database. The data should be resilient enough that even in the event of system failures or crashes, it remains intact. If any data is lost, the recovery manager takes responsibility to ensure the database's durability. To

commit changes to the database, the 'COMMIT' command should be used. Consequently, the ACID properties of DBMS are crucial in preserving the consistency and reliability of data.

Concurrency Control

Distributed Concurrency Control (DCC) refers to the coordination and management of simultaneous operations in a distributed database system. In distributed systems, data may reside on multiple servers or nodes, and clients from different locations may want to access or modify this data at the same time. Managing this concurrent access is crucial to maintaining the consistency and reliability of the data.

A robust concurrency control mechanism for distributed DBMSs should:

- Be resilient to both site and communication failures.
- Allow parallelism to meet performance requirements.
- Incur minimal computational and storage overhead.
- Perform well even in network environments with significant communication delays.
- Impose minimal restrictions on the structure of atomic actions.

Concurrent access by multiple users to a database can lead to issues such as lost updates, uncommitted dependencies, and inconsistent analyses. These problems are present in distributed settings and are exacerbated by the intricacies of data distribution. One prominent challenge is the multiple-copy consistency problem. This issue arises when a single data item exists in multiple locations. To ensure consistency across the global database, all replicas of an item must be updated when one site modifies it. Failure to update all copies results in inconsistencies within the database.

Issues with Concurrent Execution:

Database transactions primarily involve two operations: READ and WRITE. Managing these operations during concurrent execution of transactions is crucial, as improper interleaving can lead to data inconsistencies. Consequently, several problems can emerge:

1. **Lost Update Problems (W - W Conflict)**:
   This issue arises when two different database transactions execute read/write operations on the same database item in an interleaved fashion (concurrent execution). This can result in

incorrect values, leading to database inconsistency. E.G. Consider two transactions, T_X and T_Y, operating on the same account A with a balance of $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, TX reads the value of account A as $300.
- At time t2, TX deducts $50 from account A, resulting in $250 (not yet written).
- Concurrently, at time t3, TY reads the original value of account A, which is $300, as TX hasn't updated it.
- At time t4, TY adds $100 to account A, totalling $400 (not yet written).
- At time t6, TX writes the new value of account A as $250.
- Finally, at time t7, TY writes the balance of account A as $400, causing the $250 value from TX to be lost.

2. Dirty Read Problems (W-R Conflict):

This problem occurs when a transaction updates a database item, but due to some failure, it needs to rollback. However, before the rollback is completed, another transaction accesses the updated item.

Example: Consider two transactions, TX and TY, operating on account A with a balance of $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, TX reads account A's balance as $300.
- At time t2, TX adds $50, making the balance $350.
- At time t3, TX writes the updated balance.
- At time t4, TY reads the balance as $350.
- At time t5, TX experiences a failure and rollbacks, reverting the balance to $300. However, TY still sees the balance as $350, leading to a dirty read.

3. Unrepeatable Read Problem (W-R Conflict):

The Inconsistent Retrievals Problem, also known as the Unrepeatable Read problem, arises when a single transaction reads two different values for the same database item during its execution.

Example: Imagine two transactions, TX and TY, both interacting with account A which has an initial balance of $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At t1, transaction TX reads the balance of account A and finds it to be $300.
- At t2, transaction TY also reads the balance of account A, which remains $300.
- At t3, transaction TY adds $100 to the account A balance, increasing it to $400.
- At t4, transaction TY commits the updated value, solidifying the balance at $400.
- Then, at t5, transaction TX reads the balance of account A again, this time getting a value of $400.

This indicates that within the span of transaction TX, it observed two different values for account A: initially $300 and later $400 due to changes made by transaction TY. This phenomenon, where a single transaction reads different values for the same item, is termed an "unrepeatable read" and underscores the Unrepeatable Read problem.

2.4.5 Concurrency Control Protocols:

Different concurrency control protocols balance various benefits, specifically the level of concurrency they permit against the overhead they introduce. The following are the concurrency control techniques in DBMS:

- Lock-Based Protocols
- Two-Phase Locking Protocol
- Timestamp-Based Protocols
- Validation-Based Protocols.

Lock-Based Protocols:

These protocols manage access to data items by locking them. Transactions request locks on data items before performing operations. If a transaction has a lock on an item, no other transaction can access it until the lock is released.

Types:

- Shared Lock (S-Lock): Allows the transaction that holds the lock to read the locked data item.
- Exclusive Lock (X-Lock): Allows the transaction that holds the lock to both read and write the locked data item.

Properties: Straightforward, widely used, but can lead to deadlocks.

Two Phase Locking Protocol (2PL):

The Two-Phase Locking Protocol, also known as the 2PL protocol, is a method of concurrency control in DBMS. It ensures serializability by applying locks to transaction data, preventing other transactions from accessing the same data simultaneously. The 2PL protocol helps eliminate concurrency problems in DBMS.

This protocol divides the execution phase of a transaction into three distinct parts:

1. In the initial phase, as the transaction begins its execution, it seeks permission for the locks it requires.
2. The second phase involves the transaction obtaining all the necessary locks. Once the transaction releases its first lock, the third phase is initiated.
3. In this final phase, the transaction cannot request any new locks. It can only release the locks it has previously acquired.

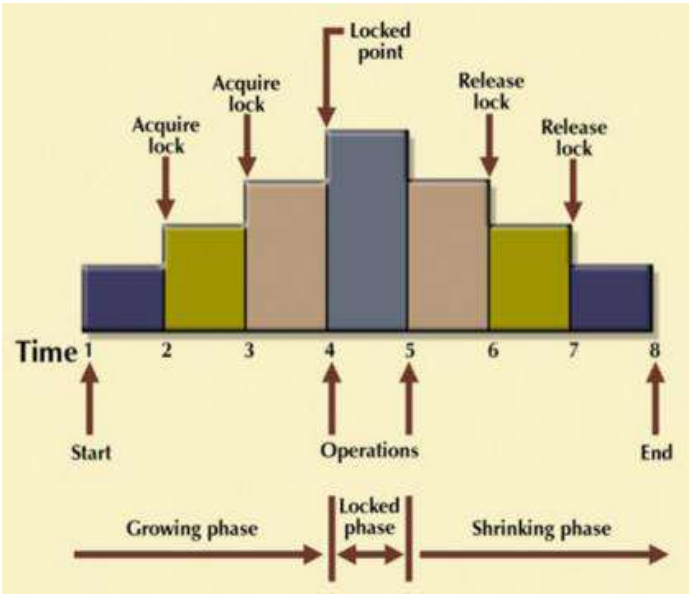


Figure 2.20 Phases of 2PL

A transaction's locking requests are divided into two phases: growing and shrinking (Figure 2.20)

- Growing Phase: A transaction may obtain locks but cannot release any.
- Shrinking Phase: A transaction may release locks but cannot acquire new ones.

Properties: Ensures conflict-serializability, but can lead to deadlocks. It ensures that once a transaction releases its locks (enters the shrinking phase), it cannot interfere with other transactions, preserving serializability.

Timestamp-Based Protocols:

Transactions are assigned a unique timestamp upon creation. The DBMS uses these timestamps to order transactions, ensuring older transactions get priority over newer ones.

Types:

- Wait-Die: Older transactions may wait if a younger transaction holds a conflicting lock. If an older transaction requests a conflicting lock, the younger transaction is aborted ("killed").
- Wound-Wait: If a younger transaction requests a conflicting lock, it's made to wait. If an older transaction requests a lock held by a younger transaction, the younger one is aborted ("wounded").

Properties: Deadlock-free by design but can lead to higher rates of transaction aborts.

Validation-Based Protocols (Optimistic Concurrency Control):

Transactions don't acquire locks during execution. Instead, they validate their changes at the end to ensure there are no conflicts.

Phases:

- Read Phase: Transaction reads data without acquiring locks.
- Validation Phase: The system checks if the transaction's changes will conflict with others.
- Write Phase: If validation succeeds, the transaction's changes are permanently written.

Properties: Offers high concurrency and avoids deadlocks but can lead to higher abort rates if many transactions conflict during validation.

Each concurrency control protocol has its advantages and trade-offs. The choice of protocol often depends on the system's specific needs, desired levels of concurrency, and tolerance for transaction aborts or restarts.

2.4.6 Distributed Database Recovery

Distributed Database Recovery is essential for maintaining data consistency and system reliability in a distributed computing environment, especially in the face of failures. Key strategies include the use of atomic commitment, employing protocols like Two-Phase Commit (2PC) or Three-Phase Commit (3PC), and implementing distributed transaction logs for coherent recovery. Techniques such as checkpointing, periodic snapshots, and quorum-based replication contribute to robust recovery mechanisms. Challenges, such as network partitions and asynchronous replication, require meticulous strategies to ensure ongoing data consistency. The choice of a recovery approach is pivotal, ensuring the resilience of the distributed database system while meeting specific infrastructure requirements for effectiveness and reliability.

In the case of failures, the Distributed Database Management System (DDBMS) takes decisive actions, including aborting affected transactions and flagging failed sites. Following a failure, the DDBMS monitors the recovery status of the affected sites, ensuring subsequent recovery procedures for maintaining database consistency. Coordinated efforts are essential during network partitions to prevent conflicting decisions on global transactions, thereby safeguarding their atomicity across partitions.

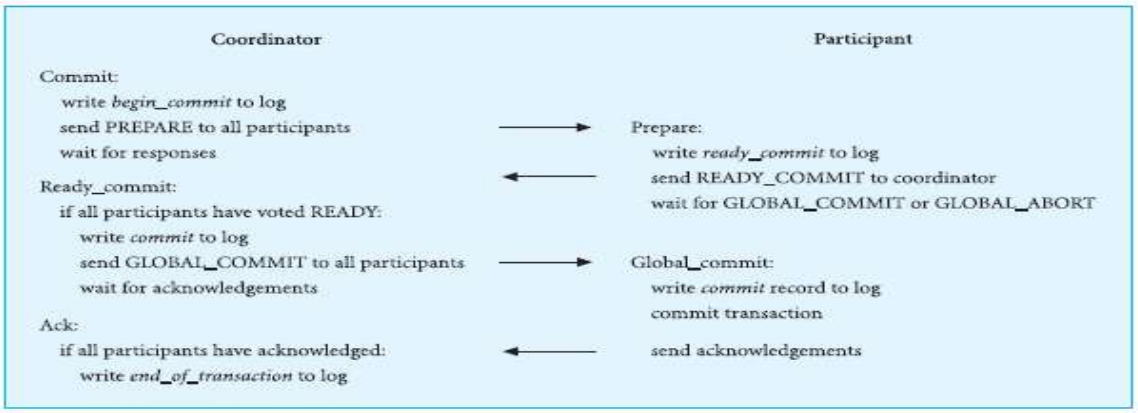
Recovery in a Distributed Database Management System is intricate, requiring the assurance of atomicity for both local sub-transactions and global transactions. Protocols like 2PC and 3PC, with a coordinator managing transactions, address these complexities, ensuring continuity even in the face of failures

Two-Phase Commit(2PC)

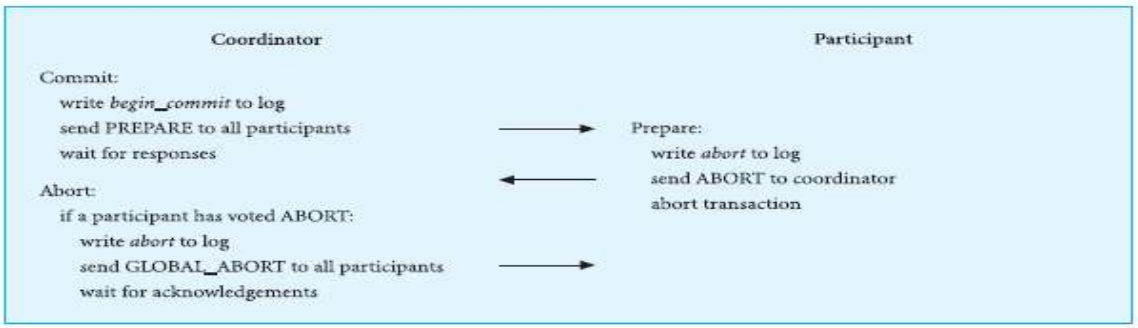
The Two-Phase Commit (2PC) protocol operates in two phases: a voting phase and a decision phase, ensuring the atomicity of distributed transactions. The coordinator initiates the process by asking participants if they are prepared to commit. If any participant votes to abort or fails to respond within a timeout, the coordinator instructs all participants to abort. If all participants

vote to commit, the coordinator instructs a global commit. Participants can unilaterally abort until they vote to commit. The protocol employs timeouts to prevent unnecessary blocking, and the coordinator waits for votes from all participants. In the commit phase, the coordinator initiates a sequence involving log writes, acknowledgment waits, and global messages to ensure a consistent global decision.

For participants, the commitment process involves responding to prepare messages, making ready_commit or abort log entries, and acknowledging global commit or abort messages. If a participant fails to receive instructions or the coordinator fails to receive a response, a termination protocol is triggered, assuming site failure. Operational sites follow the termination protocol, while failed sites undergo recovery upon restart. The 2PC protocol relies on synchronized communication, timeouts, and acknowledgment mechanisms to maintain transactional integrity in distributed environments.



(a)



(b)

Figure 2.21 Summary of 2PC: (a) 2PC protocol for participant voting COMMIT; (b) 2PC protocol for participant voting ABORT

Termination Protocol for 2PC

A termination protocol is triggered when either a coordinator or participant fails to receive an expected message within the specified timeout. The actions taken depend on the role and timing of the timeout. For the coordinator, which can be in INITIAL, WAITING, DECIDED, or COMPLETED states, timeouts during WAITING lead to global abortion decisions, and timeouts during DECIDED prompt the retransmission of global decisions to non-acknowledging sites..

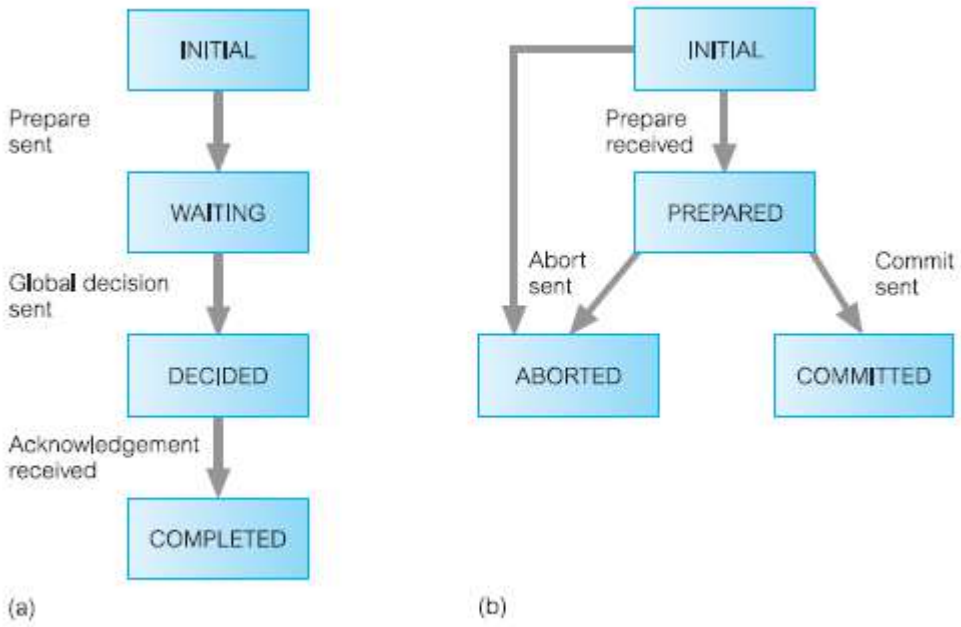


Figure 2.22 State Transition diagram for 2PC: (a) Coordinator; (b) Participant

Participants, in INITIAL, PREPARED, ABORTED, or COMMITTED states, may only experience timeouts in the first two states. In INITIAL, a participant awaits a PREPARE message, and upon timeout, can unilaterally abort the transaction. In the PREPARED state, the participant, having voted to commit, may employ a cooperative termination protocol to contact others and determine the global decision. While cooperative termination reduces the likelihood of blocking, occasional blocking may still occur, and the blocked process must persistently attempt to unblock as failures are repaired. If only the coordinator fails, participants can elect a new coordinator to resolve the block..

Recovery protocols for 2PC

Recovery actions for operational and failed sites in a distributed system vary based on the stage of the coordinator or participant at the time of failure.

Coordinator Failure:

1. Failure in INITIAL State:
   - The coordinator hasn't initiated the commit procedure.
   - Recovery involves starting the commit procedure on restart.
2. Failure in WAITING State:
   - The coordinator has sent PREPARE messages but hasn't received all responses.
   - Recovery restarts the commit procedure.
3. Failure in DECIDED State:
   - The coordinator has instructed participants to globally abort or commit.
   - On restart, if the coordinator received all acknowledgments, it completes successfully; otherwise, it initiates the termination protocol.

Participant Failure:

1. Failure in INITIAL State:
   - The participant hasn't voted on the transaction.
   - On recovery, the participant unilaterally aborts the transaction, recognizing that the coordinator couldn't have reached a global commit without its vote.
2. Failure in PREPARED State:
   - The participant has sent its vote to the coordinator.
   - Recovery involves using the termination protocol.
3. Failure in ABORTED/COMMITTED States:
   - The participant has completed the transaction.
   - On restart, no further action is needed.

The recovery protocol aims to ensure consistent actions across participants and independence in restarting, minimizing the need for coordination with other participants or the coordinator. This ensures the system can recover gracefully, even after failures at various stages of the distributed transaction process.

Three-Phase Commit(3PC)

The Two-Phase Commit (2PC) protocol, while widely used, is not a non-blocking protocol as it can lead to blocking in certain circumstances. For instance, a process may timeout after voting commit but before receiving the global instruction from the coordinator. To address this limitation, the Three-Phase Commit (3PC) protocol was proposed by Skeen in 1981. Unlike 2PC, 3PC is non-blocking for site failures, except in the rare event of the failure of all sites.

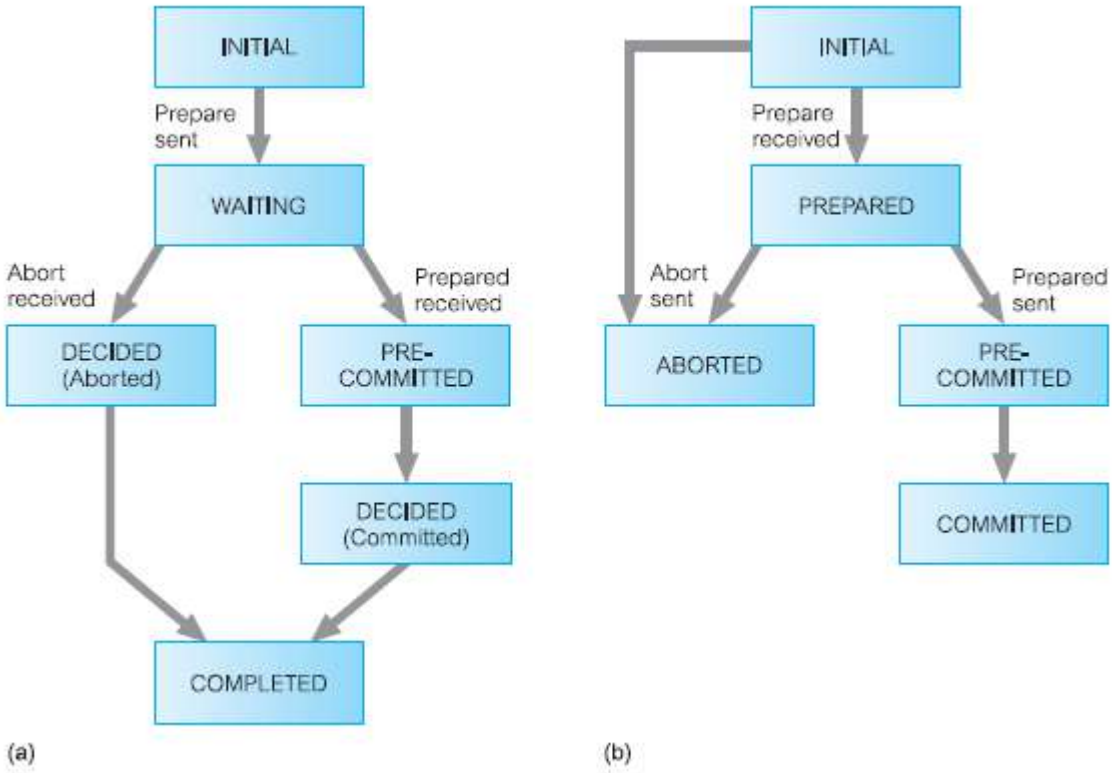


Figure 2.23: State Transition diagram for 3PC; (A) Coordinator; (b) Participant

The 3PC protocol introduces a third phase, called pre-commit, between voting and the global decision. This eliminates the uncertainty period for participants that have voted COMMIT and are waiting for the global abort or commit from the coordinator. The coordinator sends a global PRE-COMMIT message upon receiving all votes from participants. Participants acknowledge receipt of this message, and once the coordinator has received all acknowledgments, it issues the global commit. The key requirements for 3PC are that no network partitioning should occur, at least one site must always be available, and at most K sites can fail simultaneously

(classified as K-resilient). All operational processes are informed of the global decision to commit by the PRE-COMMIT message before the first process commits, allowing them to act independently in the event of failure as depicted in Figure 2.23.

If the coordinator fails, operational sites can communicate and decide whether to commit or abort without waiting for the coordinator to recover. If none of the operational sites have received a PRE-COMMIT message, they will abort the transaction. The termination and recovery protocols for 3PC ensure the protocol's robustness and non-blocking characteristics in the face of failures, providing a more reliable approach for distributed transaction coordination, as shown in Figure 2.24.

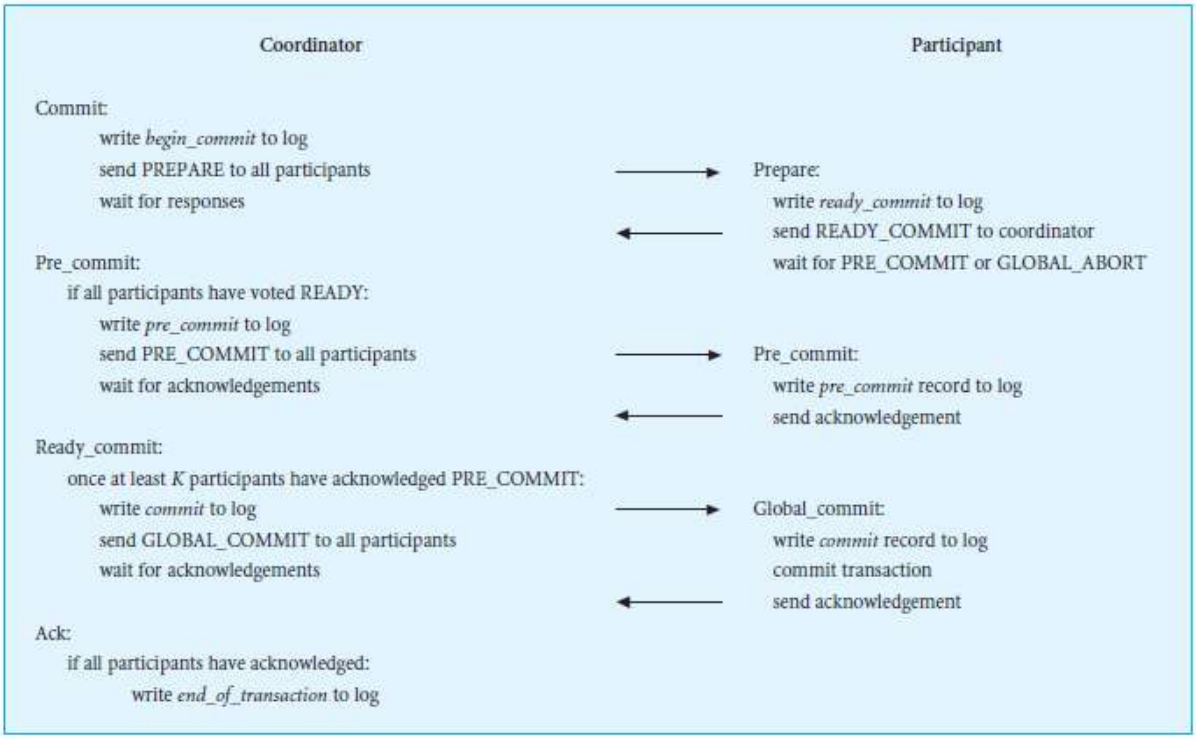


Figure 2.24: 3PC protocol for participant voting COMMIT

Termination protocols for 3PC

Termination protocols for the Three-Phase Commit (3PC) protocol involve specific actions based on the state of the coordinator or participant at the time of a timeout.

Coordinator:

1. Timeout in the WAITING State:
   - Coordinator waits for participant acknowledgments.
   - Similar to 2PC, the coordinator can decide to globally abort the transaction.
2. Timeout in the PRE-COMMITTED State:
   - Participants have received the PRE-COMMIT message.
   - Coordinator completes the transaction by writing the commit record to the log file and sending the GLOBAL-COMMIT message to participants.
3. Timeout in the DECIDED State:
   - Coordinator waits for participant acknowledgments.
   - Similar to 2PC, the coordinator sends the global decision to sites that have not acknowledged.

Participant:

1. Timeout in the INITIAL State:
   - Participant waits for the PREPARE message.
   - Similar to 2PC, the participant can unilaterally abort the transaction.
2. Timeout in the PREPARED State:
   - Participant has sent its vote to the coordinator.
   - The participant initiates an election protocol to elect a new coordinator for the transaction and terminates.
3. Timeout in the PRE-COMMITTED State:
   - Participant has sent the acknowledgment to the PRE-COMMIT message.
   - Similar to the PREPARED state, the participant initiates an election protocol to elect a new coordinator for the transaction and terminates.

These termination protocols ensure proper handling of timeouts in different states, allowing for the resolution of transactions and the election of new coordinators if necessary. They contribute to the robustness and reliability of the 3PC protocol in distributed transaction coordination.

Recovery protocols for 3PC

Recovery protocols for the Three-Phase Commit (3PC) protocol address actions to be taken upon restart, depending on the state of the coordinator or participant at the time of failure.

Coordinator Failure:

1. Failure in the INITIAL State:
   - Coordinator hasn't started the commit procedure.
   - Recovery involves initiating the commit procedure.
2. Failure in the WAITING State:
   - Participants may have elected a new coordinator and terminated the transaction.
   - On restart, the coordinator contacts other sites to determine the fate of the transaction.
3. Failure in the PRE-COMMITTED State:
   - Similar to the WAITING state, participants may have elected a new coordinator.
   - On restart, the coordinator contacts other sites to determine the fate of the transaction.
4. Failure in the DECIDED State:
   - Coordinator has instructed participants to globally abort or commit.
   - On restart, if the coordinator received all acknowledgments, it can complete successfully; otherwise, it initiates the termination protocol.

Participant Failure:

1. Failure in the INITIAL State:
   - Participant hasn't voted on the transaction.
   - On recovery, the participant can unilaterally abort the transaction.
2. Failure in the PREPARED State:
   - Participant has sent its vote to the coordinator.
   - The participant contacts other sites to determine the fate of the transaction.
3. Failure in the PRE-COMMITTED State:
   - Similar to the PREPARED state, the participant contacts other sites to determine the fate of the transaction.
   -

4. Failure in the ABORTED/COMMITTED States:
    - Participant has completed the transaction.
    - On restart, no further action is necessary.

Termination Protocol Following Election of New Coordinator:

The election protocol, as discussed for 2PC, can be utilized by participants to elect a new coordinator following a timeout. The newly elected coordinator initiates a STATE-REQ message to all participants involved in the election to determine the best course for the transaction. The new coordinator follows rules such as considering abort if any participant has aborted, committing if some have committed, or aborting if all participants are uncertain. To prevent blocking, the new coordinator first sends the PRE-COMMIT message and, upon receiving acknowledgments, sends the GLOBAL-COMMIT message. This ensures the continuity of the transaction even after the failure of the original coordinator

Self-Assessment questions and Exercises

Self-Assessment Questions

1. What is the primary goal of parallel databases?
2. How does parallelism improve query performance?
3. Name a few key parallel database architectures.
4. What is I/O parallelism, and why is it important in parallel databases?
5. Differentiate between inter-query and intra-query parallelism.
6. Provide examples of situations where each type of parallelism is useful
7. Explain the difference between inter-operation and intra-operation parallelism.
8. When is inter-operation parallelism more beneficial than intra-operation parallelism, and vice versa?
9. What are the key characteristics of a distributed database architecture?
10. Discuss the advantages and challenges of using a distributed database system.
11. What are the benefits of data replication in a distributed database?
12. Explain the concept of data partitioning and its role in distributed data storage.
13. What is a distributed transaction, and why is it complex to manage?
14. Discuss the two-phase commit protocol and its role in distributed transactions.

15. How does query optimization differ in a distributed database compared to a centralized one?
16. What is query routing, and why is it important in distributed query processing.
17. Explain the importance of concurrency control in a distributed database.

Exercises:

1. Design a parallel database architecture for a large e-commerce platform, considering the benefits and challenges.
2. Develop a distributed database schema for a global social media platform, taking into account data replication and partitioning.
3. Create a scenario involving a distributed transaction and design a recovery strategy in case of failure.
4. Optimize a complex SQL query for parallel execution and evaluate its performance.

2.5 Case Studies

Parallel Databases:

Netflix uses a parallel database system to process and analyze customer viewing data. They parallelize queries to improve recommendations and content delivery. This allows them to process vast amounts of data from millions of users efficiently.

I/O Parallelism:

Google's Bigtable relies on I/O parallelism to manage and analyze massive datasets. By splitting data into tablets and using distributed storage, they achieve efficient I/O operations, enabling Google Search and other services.

Inter-Query and Intra-Query Parallelism:

Amazon uses both inter-query and intra-query parallelism in its e-commerce platform. Inter-query parallelism helps handle multiple user requests concurrently, while intra-query parallelism optimizes product search queries by parallelizing complex operations like product recommendations.

Distributed Transactions:

Banking institutions use distributed transactions. For example, when a customer transfers funds between accounts at different branches, a distributed transaction system ensures consistency and reliability.

Distributed Transaction Management Properties - Concurrency Control:

Airbnb manages concurrent booking requests from users worldwide. To ensure data consistency and avoid double bookings, they use distributed transaction management with proper concurrency control mechanisms.

2.6 Summary

Parallel databases have revolutionized the way large-scale data processing occurs by harnessing the collective power of multiple resources. At the heart of this system is I/O parallelism, which efficiently spreads data tasks across several devices to expedite read/write operations. Two distinct forms of parallelism dominate this landscape: Inter-Query, which deals with running multiple queries simultaneously, and Intra-Query, which segments a single query into smaller tasks executed concurrently. The same division principle applies to operations, resulting in Inter-Operation and Intra-Operation parallelism. Moving to distributed databases, these systems spread across various sites, yet they operate cohesively. They manage data storage across multiple locations, handle cross-database transactions, optimize queries for distributed environments, and oversee transaction coordination in the network. Central to their functionality is upholding the ACID properties to ensure data reliability and integrity. Moreover, advanced concurrency controls are embedded to guarantee that simultaneous transactions don't jeopardize the system's stability.

Keywords

Data Partitioning, Data Skew, Fault Tolerance, Inter query, Intra query, Parallel Execution, Query Optimization, Data Fragmentation, Data Replication, Commit Protocols, 2PC, Deadlock, Redundancy, Horizontal Partitioning, Vertical Partitioning

2.6.1 Further readings

1. Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems*. Pearson.
2. Özsu, M. T., & Valduriez, P. (2020). *Principles of Distributed Database Systems*. Springer.
3. Stonebraker, M., & Brown, P. (1999). *Parallel Database Systems*. The VLDB Journal.
4. DeWitt, D. J., & Gray, J. (1992). *Parallel Database Systems: The Future of High Performance Database Systems*. Communications of the ACM.
5. Graefe, G. (1990). *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys.
6. Bernstein, P. A., & Goodman, N. (1983). *Concurrency Control in Distributed Database Systems*. ACM Computing Surveys.
7. Kossmann, D. (2000). *The State of the Art in Distributed Query Processing*. ACM Computing Surveys.
8. Garcia-Molina, H., & Salem, K. (1987). *Sagas*. ACM SIGMOD Record.

UNIT 3

XML DATABASES, WEB DATABASES, ACTIVE DATABASES AND TEMPORAL DATABASES

CONTENTS

3.1 Introduction

Data management and exchange have seen the evolution of multiple technologies and paradigms. XML (eXtensible Markup Language) serves as a standard for encoding documents in both human-readable and machine-readable formats. The structural design of XML documents can be specified using DTD (Document Type Definition) or the more advanced XML Schema, which provides a

means to validate the structural integrity of XML data. XML Querying allows users to extract and manipulate data within XML documents. In the realm of web databases, Open Database Connectivity (ODBC) offers a standard interface for accessing database management systems. Meanwhile, Java Database Connectivity (JDBC) serves a similar role, but specifically for the Java environment. PHP, a widely-used web development language, offers capabilities for accessing relational databases. The dynamic world of databases is further enriched by the Event Condition Action (ECA) model, the backbone of active databases that respond to specific conditions or events. Alongside these technologies, temporal databases store data relating to time. Interpreting time in relational databases presents unique challenges and opportunities, necessitating specialized design and implementation considerations.

3.2 Learning Objectives

- Understand XML's Role and Structure: Learn the fundamentals of XML, its encoding standards for documents, and its importance in data interchange.
- Grasp XML Document Validation Methods: Familiarize with the techniques to validate the structure of XML documents using DTD and XML Schema.
- Navigate Database Connectivity Paradigms: Explore standard interfaces like Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) for accessing database management systems.
- Learn About Dynamic Database Models: Understand the Event Condition Action (ECA) model and its role in active databases.
- Study Temporal Databases and Time Interpretation: Discover the concept of temporal databases and the challenges of interpreting time in relational databases.

3.3 Overview

The landscape of data management and exchange is intricate, marked by several key components. Foremost is the XML Data Model, a framework that elegantly structures data, making it comprehensible for both machines and humans. Ensuring this data's consistency and accuracy are the DTD and XML Schema, standards that impose order by verifying that data adheres to established rules. Beyond just storage, XML Querying empowers users to actively engage with their data, extracting and reshaping it to suit their needs.

In the wider context of the internet, where data is a crucial asset, several strategies have been developed for efficient database interaction. Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) stand out as protocols that streamline the process of connecting to databases, with JDBC specially designed for Java applications. For the realm of web development, PHP offers robust solutions, simplifying the task of fetching and manipulating data stored in relational databases.

Adding a layer of sophistication are active databases, reliant on the Event Condition Action (ECA) model. These databases excel at responding to events or triggers, enhancing data handling

with dynamic and real-time capabilities. Nevertheless, the introduction of time, a central aspect of temporal databases, brings about complexity. The interpretation of time within relational databases requires careful consideration, demanding specific design and implementation strategies to ensure temporal data's coherence and utility. These elements collectively shape the complex, multifaceted world of contemporary data management and exchange.

3.4 XML Databases

XML, originating from document management like its precursor HTML, wasn't initially intended for databases. Its foundation, SGML, focused on structuring large documents. However, XML stands out by being tailored for data representation, making it invaluable for inter-application communication and data integration. When utilized this way, various database concerns emerge, such as organizing and querying XML data. This exploration delves into XML's role in data management, addressing its use in databases and data exchanges in the form of XML-formatted documents.

3.4.1 The Semi Structured Data Model

Semi structured data bridges the gap between unstructured and structured data. Unlike rigid relational databases or completely unstructured formats like videos, semi structured data, exemplified by XML, offers flexibility with some organization. Its prominence arises from several factors. First, data structures might be inherent but not explicitly outlined. Second, integrating diverse data sources often requires a flexible model, making rigid structures unsuitable. Lastly, querying structured databases mandates schema knowledge, limiting its scope. Semi structured models, however, permit queries without complete schema awareness. This versatility positions semi structured data models, especially XML, as pivotal in managing modern data complexities.

Semi structured data models predominantly use labelled graphs for representation. In these graphs:

1. Nodes: They act as data placeholders. Each node could embody simple entities, like a date, or complex ones, such as a book.
2. Edges: Linking nodes, edges define relationships or attributes. An edge might represent that a book belongs to a specific genre or indicates an author's contribution.
3. Self-descriptive Nature: What sets semi structured data apart is its intrinsic self-descriptive character. There's no need for an external schema; the structure and relationships of the data are embedded within the graph,, eliminating the need for predefined structures.

Illustrating with the provided example, Figure 3.1 would depict a central "BOOKLIST" node. From it, three "BOOK" labeled edges would branch out, each leading to a unique book node. This configuration not only encapsulates data but also visually maps out relationships, making the data's intricacies easily comprehensible.

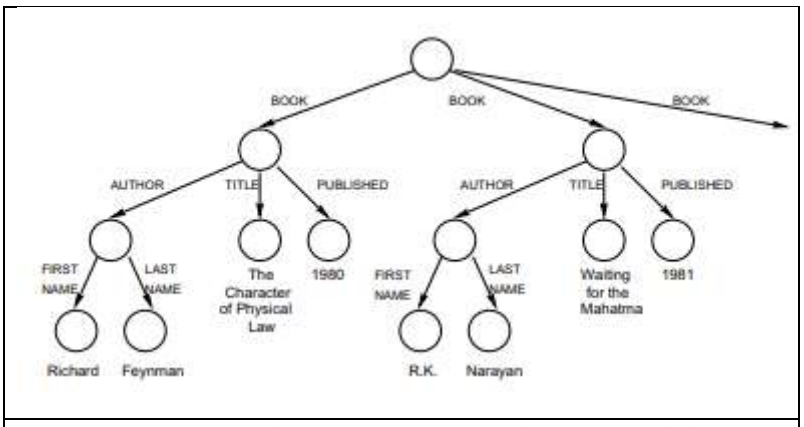


Figure 3.1 The Semi structured Data Model

3.4.2 XML Data Model

The XML Data Model is a conceptual framework for representing and accessing XML data. At its core, XML (eXtensible Markup Language) is a markup language designed to store structured data in a text format, primarily for data interchange purposes. The data model for XML is inherently hierarchical and can represent complex relationships and nested structures.
Key concepts of the XML data model include:

1. **Elements and Attributes**: XML documents consist of elements, which can have attributes. Elements can nest within other elements, creating a tree-like structure.
2. **Tree Structure**: The inherent structure of an XML document is often visualized as a tree. The topmost element, called the root, branches out to child elements, which can, in turn, have their own child elements, and so on.
3. **Text Content**: Elements can contain textual data, which is where the actual content resides in the XML structure.
4. **Namespaces**: These allow for distinguishing XML elements and attributes that may have the same name but come from different sources or contexts.
5. **Well-formedness**: A fundamental requirement for XML documents; they must adhere to specific syntactical rules to be considered "well-formed".
6. **Validation**: Beyond being well-formed, XML documents can be validated against certain rules or structures defined by DTDs (Document Type Definitions) or XML Schemas.

The XML data model's hierarchical nature is particularly well-suited for representing relationships and nested data structures, which is why XML is often chosen for configuration files, data interchange formats, and other applications requiring structured document representation.

For example, consider the following XML fragment:

```
<BOOK>
        <AUTHOR>
<FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
        </AUTHOR>
</BOOK>
```

The element AUTHOR is completely nested inside the element BOOK, and both the elements LASTNAME and FIRSTNAME are nested inside the element AUTHOR.

XML elements, or "tags", are essential components of XML documents. They structure and categorize data, ensuring clarity and organization. Each element consists of start and end tags, like **<BOOK>** and **</BOOK>**. It's vital to remember that XML is case-sensitive; hence, **<BOOK>** is different from **<book>**. Proper nesting is crucial: if one element starts inside another, it must end before the outer tag closes. Consistent application of these guidelines is vital to guarantee accurate parsing and interpretation of XML data.

Attributes in XML elements offer added details about the element. Placed within the start tag, they assign specific data to elements. For instance, **<ELM att="value">** assigns "value" to the attribute "att" of the "ELM" element. Values should be in quotes. In an example, the "BOOK" element has attributes like "genre" (e.g., science) and "format" (e.g., hardcover).

```
<BOOKLIST>
  <BOOK genre="science" format="hardcover">
    <TITLE>Exploring the Cosmos</TITLE>
    <AUTHOR>Dr. Jane Smith</AUTHOR>
  </BOOK>
  <BOOK genre="fiction" format="paperback">
    <TITLE>The Adventures of ChatGPT</TITLE>
    <AUTHOR>John Doe</AUTHOR>
  </BOOK>
</BOOKLIST>
```

In this XML:

- The dish name contains an **&**, which is represented as **&**.
- The term "Palak" is in quotes, represented using **"** entity references.
- The placeholder for spices uses angle brackets < and >, which are represented as **<** and **>** respectively.

When parsed, this XML describes a dish from India named "Paneer & Spinach Curry" and lists its ingredients. The entity references ensure the XML remains well-formed.

Input	Output
<dish> <name>Paneer & Spinach Curry</name> <ingredients> <ingredient>Paneer (Cheese)</ingredient> <ingredient> Spinach (also called "Palak "in Hindi) </ingredient> <ingredient> Spices <as per taste> </ingredient> </ingredients> <origin>India</origin> </dish>	**Dish:** Paneer & Spinach Curry **Ingredients:** • Paneer (Cheese) • Spinach (also called "Palak" in Hindi) • Spices <as per taste> **Origin:** India

Here, the entity references are converted into their actual characters:

- **&** becomes **&**
- **"** becomes **"**
- **<** becomes **<**
- **>** would become **>**, though it wasn't used in this example.

In XML, comments are initiated with <!-- and terminated with -->.

However, there's a slight correction in your statement: comments in XML start with <!-- (three characters) and end with -->.

```
<!-- This is a comment in an XML document. -->
<book>
   <title>The Art of Programming</title>
   <!-- Note: This book is recommended for advanced learners. -->
   <author>Jane Doe</author>
</book>
```

#PCDATA (Parsed Character Data):

#PCDATA represents parsed character data, which is the default content type for elements in an XML document. Parsed character data is the text found between the start tag and the end tag of an XML element. The XML parser processes this data, interpreting tags and expanding entities.

For Example:

<element_name>This is some #PCDATA text.</element_name>

In this XML instance, the term "#PCDATA" itself is merely a string of characters within the text content. It doesn't have any special meaning in this context. The use of "#PCDATA" becomes more significant when defining Document Type Definitions (DTDs) to specify the content model of an element, indicating that the content is parsed character data. In your XML instance, it's simply treated as part of the text content of the element.

CDATA (Character Data):

CDATA stands for Character Data and is used to indicate that the enclosed text should not be parsed by the XML parser. CDATA sections are often used when the content includes characters that might be misinterpreted as XML markup, such as < or &. Tags inside CDATA are treated as literal text, and entities are not expanded.

For Example:

<element_name><![CDATA[This is some <unparsed> text & entities.]] > </element_name>

CDATA sections are used when you want to include characters that would otherwise be treated as markup. In this case, the text inside the CDATA section contains <unparsed> and & entities., and these characters will not be treated as XML markup or entities by the XML parser.

Thus, the main difference between #PCDATA and CDATA lies in how the XML parser treats the content. #PCDATA is parsed by the XML processor, while CDATA is not parsed, and its content is treated as literal character data. The choice between them depends on whether you want the content to be subject to XML parsing rules or treated as plain text.

3.4.3 Document Type Declarations (DTDs)

DTD play a significant role in XML, serving as its schema language. A DTD provides a way to describe the structure of an XML document by defining:

1. Elements (tags): Specifies which elements can appear in an XML document.
2. Attributes: Designates which attributes can be associated with a specific element.
3. Entities: Enables the definition of shortcuts for long or common text strings or special characters.

By doing so, a DTD enforces a certain structure, ensuring that XML documents adhere to a specific format or blueprint. Once an XML document is associated with a DTD, any deviation from the DTD-defined structure will cause the XML document to be considered "not well-formed" or "invalid".

An Example DTD for a book:

```
<!ELEMENT book (title, author)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

This DTD indicates:

- A book must contain a title followed by an author.
- Both title and author elements can contain parsed character data (#PCDATA).

With this DTD, the following XML would be valid:

```
<book>
  <title>The Adventures of ChatGPT</title>
  <author>John Doe</author>
</book>
```

But, this would be invalid:

```
<book>
  <author>John Doe</author>
  <title>The Adventures of ChatGPT</title>
</book>
```

DTDs offer a foundational way to enforce structure on XML documents. However, as XML evolved, more sophisticated schema languages like XML Schema (XSD) were introduced, providing more features and better flexibility than DTDs.

```
<!DOCTYPE BOOKLIST [
  <!ELEMENT BOOKLIST (BOOK)*>
  <!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?)>
    <!ELEMENT AUTHOR (FIRSTNAME,LASTNAME)>
      <!ELEMENT FIRSTNAME (#PCDATA)>
      <!ELEMENT LASTNAME (#PCDATA)>
    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT PUBLISHED (#PCDATA)>
  <!ATTLIST BOOK genre (Science|Fiction) #REQUIRED>
  <!ATTLIST BOOK format (Paperback|Hardcover) "Paperback">
]>
```

Bookstore XML DTD

```
<?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE BOOKLIST SYSTEM "books.dtd">
<BOOKLIST>
<BOOK genre="Science" format="Hardcover">
  <AUTHOR>
    <FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
  </AUTHOR>
  <TITLE>The Character of Physical Law</TITLE>
  <PUBLISHED>1980</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
  <AUTHOR>
    <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
  </AUTHOR>
  <TITLE>Waiting for the Mahatma</TITLE>
  <PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
  <AUTHOR>
    <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
  </AUTHOR>
  <TITLE>The English Teacher</TITLE>
  <PUBLISHED>1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

Book Information in XML

Document Type Declarations (DTDs) offer a way to dictate the structure and content of XML documents. Encapsulated within **<!DOCTYPE name [DTDdeclaration]>**, the DTD commences with the root element, in our instance, **BOOKLIST**. The rule **<!ELEMENT BOOKLIST (BOOK)*>** signifies that **BOOKLIST** may enclose zero or multiple **BOOK** elements, as the asterisk (*****) suggests any number of occurrences. The plus symbol (+) dictates at least one occurrence, hence **<!ELEMENT BOOKLIST (BOOK)+>** ensures a minimum of one **BOOK**. Another example, **<!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?)>**, describes the **BOOK** structure, which must have an **AUTHOR** and **TITLE**, with **PUBLISHED** being optional—highlighted by the question mark (**?**). Such rules enable consistency and validation in XML documents.

3.4.4 XML Schema (XSD):

XML Schema, often referred to as XSD (XML Schema Definition), is a more advanced and expressive alternative to DTD for describing the structure and constraining the contents of XML documents. It provides richer data types and allows for more detailed constraints on XML document structure.

Some of the advantages of XSD over DTD include:

1. Written in XML: XSDs themselves are XML documents, which means you can use XML tools to edit and manage them.
2. Richer Data Typing: Beyond simple text, XSD supports various data types like integers, dates, and durations.
3. Namespaces: Supports XML namespaces, allowing for better document organization and modularity.

XML Schema (XSD) enables you to:

1. Define Elements and Attributes: Specify which elements and attributes can appear in an XML document.
2. Assign Data Types: Designate types to elements and attributes, e.g., string, integer, date, etc.
3. Set Constraints: Impose constraints, e.g., an element value's length or pattern.

Example: Suppose we want to define an XML structure for a book, which includes an author, title, and an optional publication year. Here's how we could do it:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="author" type="xs:string"/>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="published" type="xs:integer" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema (book.xsd)

```
<?xml version="1.0"?>
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="book.xsd">
<author>J.K. Rowling</author>
<title>Harry Potter and the Philosopher's Stone</title>
<published>1997</published>
</book>
```

XML Document (book.xml)

In the XML document (book.xml), note the xsi:noNamespaceSchemaLocation attribute. This attribute points to the XML Schema (book.xsd) and is used for validation. So, if you were to try and add an element not defined in the schema or provide incorrect data type (like a string for the published year), a validating XML parser would raise an error.

XML with Embedded Schema:

In the embedded schema example, the schema is part of the XML document, which might be useful for standalone documents where you want the schema and the data together.

```
<?xml version="1.0"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:schema">
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:schema">
    <xs:element name="book">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="title" type="xs:string"/>
          <xs:element name="author" type="xs:string"/>
          <xs:element name="year" type="xs:integer"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
  <book>
    <title>Embedded Schema Book</title>
    <author>Jane Doe</author>
    <year>2022</year>
  </book>
</root>
```

XML with Embedded Schema (data.xml)

XML with External Schema: The external schema example keeps the schema separate, which allows for reusability of the schema across multiple XML documents.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="author" type="xs:string"/>
        <xs:element name="year" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema (book.xsd)

XML Document referencing the external schema:

```
<?xml version="1.0"?>
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="book.xsd">
  <title>External Schema Book</title>
  <author>John Doe</author>
  <year>2023</year>
</book>
```

data.xml

When XML is processed by an application or viewed in an XML viewer, the XML Schema (XSD) is typically not shown; rather, it's used behind the scenes to validate the structure and content of the XML data. The output would look like the XML data content without the schema.

Based on the provided XML examples:

The output will be:

Embedded Schema Book

Jane Doe

2022

Remember, the actual appearance might vary based on the application used to view or process the XML. For example, if viewed in a web browser, the XML elements and their structure would be displayed. If processed by an application to display content, you might see a formatted output based on how the application is designed to present the XML content.

3.4.5 XML QUERYING

XML querying refers to the process of extracting, filtering, and manipulating information from XML documents using specific query languages. XML (Extensible Markup Language) is a standardized format used to encode structured information in both human-readable and machine-readable forms. Given the extensive use of XML in various applications—from configuration files and data storage to web services and data interchange—there is often a need to query these documents to extract meaningful information.

Several tools and languages have been developed to support XML querying:

1. XPath: A language that allows navigation of XML documents. XPath provides a way to select specific nodes in an XML tree structure. It's a foundational technology for other XML-related specifications.
2. XQuery: Built on XPath, XQuery is a comprehensive language designed for extracting information from XML data and was developed by the W3C. It allows users to formulate more complex queries, including joining data, filtering it, and constructing new XML. XQuery uses the FLWOR (For, Let, Where, Order by, Return) expression, which provides a flexible way to describe data retrieval and transformation.
3. XSLT: Although primarily a stylesheet language for XML, XSLT (Extensible Stylesheet Language Transformations) can also be used to query XML. It transforms XML from one format to another and leverages XPath for its template matching.

Usage Contexts for XML Querying:

- Data Retrieval: Extracting specific data from XML documents.
- Data Transformation: Modifying the structure or content of XML data, like converting XML data into other formats (e.g., XML to HTML or XML to JSON).
- Data Integration: Combining data from multiple XML sources or even mixing XML with other data forms.
- Web Services: Processing XML payloads in web services, especially in SOAP-based services where the message format is XML.
- XML Databases: There are databases designed specifically for XML data (e.g., BaseX, eXist-db, MarkLogic). Querying these databases often requires specialized XML query languages.

XPath (XML Path Language)

XPath is a powerful language used for navigating through and selecting nodes from an XML document. It is both a standalone language and a core component of XSLT, XQuery, and XML Schema.

Usage of XPath:

1. Node Selection: At its core, XPath allows you to traverse the XML structure and select specific nodes based on various criteria.
2. XML Document Navigation: XPath provides a way to move up and down the XML hierarchy, selecting parents, siblings, or children of a node.
3. Conditional Selection: It allows for the conditional selection of nodes based on their attributes, values, or positions.
4. String, Number and Date Functions: XPath comes equipped with built-in functions to operate on string values, numbers, and dates.
5. Integration with Other XML Technologies: XPath is utilized as a foundational technology in several other XML-related specifications, such as XSLT (for XML transformations) and XQuery (for XML querying).
6. Web Scraping and Testing: XPath is often used in web scraping to extract information from web pages, and in testing frameworks to verify the content of XML or HTML documents.

Example:

Given the XML document: If you want to retrieve the names of all employees using XPath

Example 1:

<employees> <employee id="101"> <name>John Doe</name> <position>Manager</position> </employee> <employee id="102"> <name>Jane Smith</name> <position>Engineer</position> </employee> </employees>	XML File
/employees/employee/name	XPath Query

Example 2:

<library> <book id="1"> <title>Harry Potter and the Philosopher's Stone</title> <author>J.K. Rowling</author> <published>1997</published> </book> <book id="2"> <title>The Hobbit</title> <author>J.R.R. Tolkien</author> <published>1937</published> </book> </library>	XML File
XPath Query/Output	
/library/book/title	Select all book titles

<title>Harry Potter and the Philosopher's Stone</title> <title>The Hobbit</title>	

/library/book[@id='2']/author	Select the author of the book with id=2:
<author>J.R.R. Tolkien</author>	
/library/book[published < 1950]	
<book id="2"> <title>The Hobbit</title> <author>J.R.R. Tolkien</author> <published>1937</published> </book>	Select books published before 1950
/library/book[1]/title	
<title>Harry Potter and the Philosopher's Stone</title>	Select the author of the book with id=2:
count(/library/book)	Count the number of books

XPath offers a compact and expressive syntax to pinpoint the exact parts of an XML document we're interested in. When paired with other XML tools or programming languages, it becomes an invaluable tool for processing XML data.

XPath is essential in XSLT, as it's used to match patterns of nodes, navigate trees, and retrieve values. XSLT (eXtensible Stylesheet Language Transformations) allows you to transform XML data from one format to another, which might be another XML, HTML, or any text format.

Here's a basic example to illustrate how XPath is used within XSLT:

```
<library>
   <book id="1">
      <title>Effective Java</title>
      <author>Joshua Bloch</author>
   </book>
   <book id="2">
      <title>Introduction to Algorithms</title>
      <author>Thomas H. Cormen</author>
   </book>
</library>
```

books.xml

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Template for the entire XML document -->
  <xsl:template match="/">
    <html>
      <body>
        <h2>Book List</h2>
        <table border="1">
          <tr>
            <th>Title</th>
            <th>Author</th>
          </tr>
          <!-- Apply templates for each book element -->
          <xsl:apply-templates select="library/book"/>
        </table>
      </body>
    </html>
  </xsl:template>
  <!-- Template for each book element -->
  <xsl:template match="book">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="author"/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

transform.xsl

```
<html>
  <body>
    <h2>Book List</h2>
    <table border="1">
      <tr>
        <th>Title</th>
        <th>Author</th>
      </tr>
      <tr>
        <td>Effective Java</td>
        <td>Joshua Bloch</td>
      </tr>
      <tr>
        <td>Introduction to Algorithms</td>
        <td>Thomas H. Cormen</td>
      </tr>
    </table>
  </body>
</html>
```

Output

Explanation:

- **<xsl:template match="/">** : This template rule matches the root of the XML document. The content inside this rule will be applied for the entire XML.
- **<xsl:apply-templates select="library/book"/>** : Here, XPath is used to select all **book** elements inside **library**. It then applies the template rules for each **book** element.
- **<xsl:template match="book">** : This template rule matches each **book** element. For each **book**, it creates a new row in the table.

- **<xsl:value-of select="title"/> and <xsl:value-of select="author"/>**: These are XPath expressions that retrieve the text values of the **title** and **author** elements, respectively.

When you apply this XSLT to the provided XML, you'll get an HTML table listing all the books with their titles and authors.

This output is an HTML representation of the XML data. When viewed in a web browser, it will display a table titled "Book List" with two rows, each corresponding to a book from the original XML document. The table will list the titles and authors of the books.

XQuery

XQuery is a functional query language specifically designed to query XML data. It's a robust and versatile language that allows you to extract and manipulate data from XML documents or any other data source that can be viewed as XML, such as relational databases or web services.

Key Features of XQuery:

1. Functional Language: XQuery is functional in nature, meaning functions are first-class citizens, and it uses expressions rather than statements.
2. FLWOR Expressions: One of the primary constructs in XQuery. The acronym stands for "For, Let, Where, Order by, Return", which represents the main components of the expression.
3. Rich Set of Built-in Functions: XQuery has a plethora of built-in functions to handle strings, numbers, dates, sequences, and more.
4. Path Expressions: Much like XPath, you can navigate XML structures.
5. Support for Namespaces: Helpful when working with XML documents that utilize namespaces.
6. Type Awareness: XQuery is aware of XML Schema types.
7. Ability to Construct XML: Apart from querying, you can construct new XML fragments or documents.

XQuery Usage:

1. Data Retrieval: Extract specific elements or attributes from an XML document.
2. Data Transformation: Convert XML data into another XML structure, HTML, plain text, or other formats.
3. Filtering Data: Fetch data based on specific conditions.
4. Combining Data: Join multiple XML datasets.
5. Generating Summaries: Produce summaries or aggregated data from XML datasets.
6. Text Search: Implement text search functionalities over XML data.

Usage Contexts:

1. XML Databases: There are databases (e.g., eXist-db, BaseX, MarkLogic) designed specifically to store XML documents. XQuery is typically the primary means to query these databases.
2. Data Integration: When working with multiple XML sources or combining XML with other data forms, XQuery can be instrumental.
3. Web Services: With many web services using XML as a format (especially SOAP-based services), XQuery can be employed to process the XML payloads.
4. Document Transformations: XQuery can be used to convert XML documents into other formats, including other XML structures, HTML, or plain text.

Simple Example:

Consider an XML document named **library.xml**:

```
<library>
  <book id="1">
    <title>Effective Java</title>
    <author>Joshua Bloch</author>
    <published>2008</published>
  </book>
  <book id="2">
    <title>Introduction to Algorithms</title>
    <author>Thomas H. Cormen</author>
    <published>2009</published>
  </book>
</library>
```

If you want to extract the titles of all books published after 2008, you would use the following XQuery:

```
declare variable $doc := doc("library.xml");
for $book in $doc/library/book
```

where $book/published > 2008

return $book/title

When this XQuery is executed, the output will be:

<title>Introduction to Algorithms</title>

XQuery is primarily designed for querying XML data, and it doesn't have built-in join operations like those found in relational databases. However, you can perform similar operations using XQuery's capabilities for navigating and selecting nodes in XML documents.

Let's consider an example with two XML documents: one representing a list of books and another representing a list of authors. We want to retrieve information about books along with their respective authors.

books.xml	author.xml
<books> <book> <title>Introduction to XQuery</title> <authorId>1</authorId> </book> <book> <title>Database Systems</title> <authorId>2</authorId> </book> </books>	<authors> <author> <id>1</id> <name>John Doe</name> </author> <author> <id>2</id> <name>Jane Smith</name> </author> </authors>

Now, XQuery can be used to join these documents and retrieve information about books along with their authors.

XQuery	Output

let $books := doc("books.xml")//book
let $authors := doc("authors.xml")//author
for $book in $books
let $authorId := $book/authorId
	<result>
	<title>Introduction to XQuery</title>
	<author>John Doe</author>
	</result>

let $author := $authors[id = $authorId] return <result> <title>{data($book/title)}</title> <author>{data($author/name)}</author> </result>	<result> <title>Database Systems</title> <author>Jane Smith</author> </result>

The provided XQuery code retrieves information about books along with their authors by joining two XML documents ("books.xml" and "authors.xml"). Based on the given XML data,
This output represents a sequence of <result> elements, each containing the title of a book and the corresponding author's name. The XQuery code iterates through the books, matches the author IDs, and retrieves the relevant information from the "authors.xml" document.

Running the XQuery:

To execute XQuery, you would typically use an XQuery processor or an XML database that supports XQuery. There are several XQuery processors available, including:

- BaseX
- eXist-db
- MarkLogic
- Saxon

For the above example, if using BaseX:

1. Store the XML content in library.xml.
2. Save the XQuery code into a file, say query.xq.
3. Run the query using BaseX.

The method to execute the XQuery might vary based on the tool or environment you're using. In essence, XQuery is a powerful tool for anyone working with XML data, allowing for complex queries, data extraction, and transformation.

3.5 Web Databases

A web database is essentially a database that is accessed and interacted with over the Internet. It provides a way to persistently store, manage, and retrieve data for web applications. Given the prevalence of web-based applications in today's digital age, web databases have become fundamental building blocks for many services and platforms.

Characteristics of a Web Database:

1. **Accessibility:** Web databases can be accessed from anywhere via the Internet, allowing for decentralized and distributed data access.
2. **Concurrent Users:** They are designed to handle multiple simultaneous users or requests, which is a standard requirement for web applications.
3. **Security:** Given that these databases are accessible via the web, they implement robust security measures, including encryption, authentication, and authorization mechanisms.
4. **Scalability:** Web databases often support scalability both vertically (adding more power to the existing server) and horizontally (adding more servers to the network) to handle increased loads.
5. **Integration with Web Technologies:** These databases can be seamlessly integrated with web technologies and platforms, such as web servers, RESTful APIs, and more.

Types of Web Databases:

1. **Relational Databases:** These use tables to store data and are based on the relational model. Examples include MySQL, PostgreSQL, and Microsoft SQL Server.
2. **NoSQL Databases:** These are non-relational databases optimized for specific data models and have flexible schema. Types of NoSQL databases include:
    - **Document-Based:** MongoDB, CouchDB
    - **Key-Value Stores:** Redis, Riak
    - **Column Stores:** Cassandra, HBase
    - **Graph-Based:** Neo4j, OrientDB
3. **NewSQL Databases:** These are modern relational databases that aim to provide the scalability of NoSQL systems while maintaining the ACID guarantees of a traditional RDBMS. Examples include Google Spanner and CockroachDB.
4. **Cloud Databases:** These are databases hosted on cloud platforms, offering scalability, flexibility, and often a pay-as-you-go model. Examples include Amazon RDS, Azure SQL Database, and Google Cloud SQL.

Applications of Web Databases:

1. **E-Commerce:** For storing product information, user profiles, transactions, etc.
2. **Content Management Systems:** For storing articles, images, and user data.
3. **Social Networks:** For storing user profiles, friend lists, posts, and more.
4. **Analytics Platforms:** For storing and querying large volumes of data to derive insights.
5. **Web Services and APIs:** To provide data endpoints for other applications or services.

Considerations for Using a Web Database:

1. **Backup and Recovery:** Regular backups are crucial to prevent data loss.
2. **Performance Tuning:** Optimizing the database to handle web traffic efficiently.
3. **Security Measures:** Protecting against threats like SQL injection, unauthorized access, etc.
4. **Data Privacy:** Ensuring compliance with data privacy regulations like GDPR or CCPA.

Accessing SQL Through Programming Languages

Modern software applications often require interactions with relational databases for tasks like data storage and retrieval. Typically, these interactions are facilitated using SQL (Structured Query Language). However, rather than directly embedding SQL commands within the source code of a programming language, developers leverage standardized APIs such as ODBC and JDBC.

These APIs furnish a comprehensive set of functionalities that facilitate tasks like connecting to the database, submitting SQL queries, and managing the responses. One of the primary advantages of using these APIs is the abstraction layer they introduce. By doing so, they allow developers to remain focused on building the core logic of their application without getting mired in the intricacies of database-specific nuances and SQL command details.

ODBC and JDBC, in their roles, act as connectors, linking the domain of programming with that of databases. This ensures that data can be smoothly exchanged, manipulated, and stored between software applications and their backend relational databases.

3.5.1 Open Databases Connectivity

Open Database Connectivity (ODBC) is an API that standardizes access to database management systems (DBMS). It provides a way for client applications to communicate and interact with any database for which there's an ODBC driver available, without having to know specifics about the DBMS. Every ODBC-compatible database system provides a dedicated library. When an

application makes an ODBC API request, the library's code communicates with the database server, executes the desired action, and gathers the results.

The C code example illustrates the ODBC API's utilization:

```
#include <sql.h>
#include <sqlext.h>
#include <stdio.h>

void ODBCExample() {
    RETCODE rc;
    HENV environment;
    HDBC dbConnection;
    HSTMT statement;

    // Setup: Allocating environment and connection
    SQLAllocEnv(&environment);
    SQLAllocConnect(environment, &dbConnection);

    // Connecting to the database
    SQLConnect(dbConnection, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
"avipasswd", SQL_NTS);

    // Executing SQL query and binding results
    char department[80];
    float avgSalary;
    int lenDept, lenSalary;
```

```
    char * query = "SELECT dept_name, AVG(salary) FROM instructor GROUP BY
dept_name";
    SQLAllocStmt(dbConnection, &statement);
    rc = SQLExecDirect(statement, query, SQL_NTS);

    if (rc == SQL_SUCCESS) {
        SQLBindCol(statement, 1, SQL_C_CHAR, department, 80, &lenDept);
        SQLBindCol(statement, 2, SQL_C_FLOAT, &avgSalary, 0, &lenSalary);

        while (SQLFetch(statement) == SQL_SUCCESS) {
            printf("%s %f\n", department, avgSalary);
        }
    }
    // Cleanup
    SQLFreeStmt(statement, SQL_DROP);
    SQLDisconnect(dbConnection);
    SQLFreeConnect(dbConnection);
    SQLFreeEnv(environment);
}
```

1. To establish a connection to a server, an SQL environment is set up initially, followed by a database connection handle.
2. The application then opens a database connection using the **SQLConnect** function, inputting parameters like the connection handle, server name, user ID, and password.
3. After setting up the connection, SQL commands can be dispatched to the database through the **SQLExecDirect** function.
4. C language variables can be linked to the resulting query attributes. So, when the **SQLFetch** function retrieves a result tuple, its attribute values get placed into the relevant C variables. The **SQLBindCol** function manages this binding process.

5. The **SQLFetch** function is looped to retrieve all the results. With every iteration, the program fetches values and stores them in the defined C variables.
6. Once the session concludes, the statement handle is released, the database connection is severed, and the SQL environment and connection handles are cleared.

Additionally, ODBC allows for parameterized SQL statements. For instance, "INSERT INTO department VALUES (?, ?, ?)" could be prepared and repeatedly executed by providing real values for the placeholders. ODBC provides a variety of functions, including retrieving details about the database's relations and the columns' names and types in a query result or database relation.

ODBC treats every SQL statement as a distinct, automatically committed transaction. Users can disable this auto-commit feature, necessitating manual transaction commits or rollbacks. The ODBC standard also defines varying conformance levels, dictating the provided functionalities. The SQL standard has a Call Level Interface (CLI) resembling the ODBC interface.

3.5.2 Java Database Connectivity

The Java Database Connectivity (JDBC) standard establishes an API that Java applications can leverage to interact with databases. The sample Java code provided demonstrates the utilization of the JDBC interface. The sequence showcased involves establishing a connection, executing statements, handling the results, and finally, closing the connection. In this process, the program must first import the necessary JDBC interface definitions from java.sql package.

```
public static void JDBCexample(String userid, String passwd)
{
try
{
Class.forName ("oracle.jdbc.driver.OracleDriver");
Connection conn = DriverManager.getConnection (
"jdbc:oracle:thin:@db.yale.edu:1521:univdb", userid, passwd);
Statement stmt = conn.createStatement();
try {
        stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
    }
catch (SQLException sqle)
{
System.out.println("Could not insert tuple. " + sqle);
}
ResultSet rset = stmt.executeQuery(
```

```
                "select dept name, avg (salary) "+
                " from instructor "+
                " group by dept name");
while (rset.next()) {
                System.out.println(rset.getString("dept name") + " " +
                rset.getFloat(2));
                    }
stmt.close();
conn.close();
}
catch (Exception sqle)
{
System.out.println("Exception : " + sqle);
}
}
```

Establishing a Database Connection in Java:

To communicate with a database using Java, a connection must be established. This involves several steps:

Loading the JDBC Driver: Before establishing a connection, the relevant JDBC driver needs to be loaded. This is done using the Class.forName method. For instance, to load the Oracle driver, you would use Class.forName("oracle.jdbc.driver.OracleDriver");. The driver is often provided by the database vendor in a .jar file.

Opening a Connection: Once the driver is loaded, a connection can be opened using the DriverManager.getConnection method. This requires a database URL (often specifying the machine name, protocol, port, and specific database), a username, and a password.

Sending SQL Statements:

After a connection is open, SQL statements can be sent to the database using a Statement object. The methods executeQuery (for queries) and executeUpdate (for non-queries like INSERT, UPDATE, DELETE) are used.

For instance:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM instructor");
```

Result of a Query:
When a query is executed, the results are retrieved into a ResultSet object. This allows you to iterate over the rows and fetch data from each column using methods like getString, getFloat, etc.

Prepared Statements:
A PreparedStatement allows SQL queries to be precompiled and reused with different values. This is useful for efficiency and to prevent SQL injection attacks. Values in the statement can be set using methods like setString, setInt, etc. and the query can then be executed multiple times with different values.

Example:

```
PreparedStatement pStmt = conn.prepareStatement("insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
```

Callable Statements:
JDBC offers the CallableStatement interface for executing stored procedures and functions in the database. These work similarly to prepared statements, but for calling procedures/functions.

Metadata Features:
JDBC provides a way to retrieve metadata, or information about the database structure and capabilities. The ResultSetMetaData object provides details about the result set of a query, like column names and types. Meanwhile, the DatabaseMetaData object provides information about the database itself, like table names, column names, and more.

Example:

```
ResultSetMetaData rsmd = rs.getMetaData();
int columns = rsmd.getColumnCount();
for(int i = 1; i <= columns; i++) {
   System.out.println(rsmd.getColumnName(i));
   System.out.println(rsmd.getColumnTypeName(i));
}
```

Java provides a comprehensive set of tools via the JDBC API to connect to, query, and interact with databases in a structured and secure manner.

3.5.3 Accessing Relational Database using PHP

PHP Hypertext Preprocessor, is an open-source, server-side scripting language widely used in web development. Unlike client-side languages, PHP runs on the server, enabling dynamic content generation. When integrated with HTML, it facilitates interactive web pages. A standout feature is its ability to interact with databases, making data-driven websites feasible. It powers major platforms like WordPress. While it faces criticisms, especially about security, adhering to best practices mitigates most issues. Supported by a vast community and myriad frameworks, PHP remains instrumental in shaping the online landscape.

Accessing a relational database using PHP generally involves these steps:

1. Establishing a connection to the database.
2. Constructing SQL queries.
3. Sending the queries to the database.
4. Processing the results.
5. Closing the connection.

Accessing a relational database using PHP can be done using various methods. Two of the most common methods are:

1. PDO (PHP Data Objects): A database access layer that provides a consistent interface for accessing various databases.
2. MySQLi (MySQL Improved): An extension specifically for MySQL databases.

Here's a brief overview and example of each:

1. **Using PDO:**

With PDO, you can connect to a variety of databases, not just MySQL. Here's a basic example of using PDO to connect to a MySQL database and fetch some data:

This PHP script uses PDO to connect to a MySQL database testdb on localhost. It defines connection parameters like host, dbname, username, and password. The script then inserts a new user record with a name and email into the users table using a prepared statement, which is safer against SQL injections. After inserting the data, it retrieves all user records from the table and displays them. Any connection errors or exceptions during database operations are caught and displayed. Finally, the connection to the database is closed by setting the connection object to null. This example showcases basic insert and fetch operations using PDO in PHP.

```
<?php

$host = 'localhost';
$dbname = 'testdb';
$user = 'username';
$password = 'password';
```

```
try {
   // 1. Connect to the database.
   $conn = new PDO("mysql:host=$host;dbname=$dbname", $user, $password);
   $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

   // 2. Insert a new record.
   $name = 'John Doe';
   $email = 'john.doe@example.com';

   $stmt = $conn->prepare("INSERT INTO users (name, email) VALUES (:name, :email)");
   $stmt->bindParam(':name', $name);
   $stmt->bindParam(':email', $email);
   $stmt->execute();

   echo "New record created successfully!<br>";

   // 3. Fetch records.
   $stmt = $conn->query("SELECT id, name, email FROM users");

   while($row = $stmt->fetch()) {
         echo "ID: " . $row["id"]. " - Name: " . $row["name"]. " - Email: " . $row["email"]. "<br>";
   }

 } catch(PDOException $e) {    echo "Error: " . $e->getMessage(); }

// 4. Close the connection.
$conn = null;
?>
```

Key points:

- We're using PDO's prepared statements to insert data. This is a good practice to avoid SQL injections.
- After inserting, we query the users table to fetch all records and print them.
- At the end of the script, we close the database connection by setting $conn to null.
- Remember to adjust the host, dbname, username, and password values to reflect your actual database settings. Always ensure that any user-provided data (like form submissions) is sanitized before use in any database operations to avoid potential security risks.

2. **Using MySQLi:**

Here's how you can connect to a MySQL database using the MySQLi extension and fetch some data:

```
<?php
$host = '127.0.0.1';
$db   = 'my_database';
$user = 'my_user';
$pass = 'my_password';

// Connect to the database
$mysqli = new mysqli($host, $user, $pass, $db);

// Check connection
if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}

$result = $mysqli->query("SELECT name, email FROM users");

while ($row = $result->fetch_assoc()) {
    echo $row['name'] . ' - ' . $row['email'] . "<br>";
}

$mysqli->close();
?>
```

This PHP script connects to a MySQL database using the mysqli extension. Connection parameters like host, database name, username, and password are defined. The code establishes a connection and checks if there's any error. If there's a connection error, the script terminates and displays the error message. If the connection is successful, it runs a SQL query to fetch the name and email of all users from the users table. The results are then looped through, and each user's name and email are displayed on separate lines in the browser. After displaying all records, the script closes the connection to the database.

In this example, we've used direct querying without prepared statements. For real-world applications, it's crucial to use prepared statements or other means to prevent SQL injection and ensure the security of your applications.

3.6 Event Condition Action Model

The **Event-Condition-Action (ECA) model** is a fundamental approach commonly employed in active database systems, rule-based systems, and event-driven systems. In essence, the model facilitates automated responses to specific events if a certain condition is met. Let's break down each of these components:

1. **Event**:
    - Refers to a specific change or occurrence in the system or environment.
    - Common examples in databases include operations like **INSERT**, **UPDATE**, or **DELETE**. In other systems, it might be user interactions like a button click, a sensor reading surpassing a threshold, etc.
2. **Condition**:
    - A predicate (or a test) that checks if the system should respond to an event.
    - It essentially acts as a filter. If the condition is satisfied (returns **true**), the action associated with it will be triggered. If not, nothing happens.
3. **Action**:
    - Refers to the response or task executed when the corresponding event happens, and the condition is satisfied.
    - In database systems, it might be another database operation or triggering a stored procedure. In event-driven software applications, it could be showing an alert, sending an email, etc.

Example:

Let's consider an online banking system:

- **Event**: When a user withdraws money from their account.
- **Condition**: If the remaining balance after the withdrawal is below $100.
- **Action**: Send an email notification to the user suggesting depositing more funds.

In this scenario, the ECA rule ensures that whenever a withdrawal happens (Event) and the balance falls below $100 (Condition), an email is dispatched to the user (Action).

Importance:

The ECA model is crucial in creating responsive and interactive systems. Here are some benefits:

- **Proactivity**: Systems can automatically react to specific changes without waiting for user input.
- **Consistency**: Guarantees certain actions will occur when predefined conditions are met, ensuring data integrity and business rules enforcement.
- **Flexibility**: ECA rules can be added, removed, or modified as business needs change, allowing systems to adapt to new requirements.

In active databases, ECA rules help in maintaining database integrity, automatically enforcing business rules, and responding to exceptional scenarios. In broader systems, ECA rules can facilitate complex event processing, automating workflows, and more.

3.6.1 Design and Implementation issues for Active databases

Triggers are automatic actions executed by the database in response to specific events. They are defined using SQL syntax and can be used to maintain data integrity, log changes, or automate repetitive tasks. Let's discuss some examples using Oracle 7 Server syntax, which will provide a clearer understanding of triggers and their functionalities:

Example 1: init_count Trigger In this example, the **init_count** trigger initializes a counter variable every time an **INSERT** statement is executed on the **Students** relation.

```
CREATE TRIGGER init_count
BEFORE INSERT ON Students /* Event specification */
DECLARE
   count INTEGER;
BEGIN /* Action to be taken */
   count := 0;
END;
```

Here, the **BEFORE INSERT** clause specifies that the trigger should be executed before an insert operation occurs on the **Students** table. When the trigger is activated, it simply sets the **count** variable to zero.

Example 2: incr_count Trigger In this subsequent example, the **incr_count** trigger increments the counter for each newly inserted student who is younger than 18.

```
CREATE TRIGGER incr_count
AFTER INSERT ON Students /* Event specification */
WHEN (new.age < 18) /* Condition to check age */
FOR EACH ROW
BEGIN /* Action to be taken */
   count := count + 1;
END;
```

Here, the **AFTER INSERT** clause specifies that the trigger should be executed after an insert

operation. The **WHEN** clause adds a condition to check if the age of the newly inserted student (denoted by **new.age**) is less than 18. If this condition holds true, the trigger action is executed, incrementing the **count** by 1. The **FOR EACH ROW** clause ensures this check and action are performed for every individual record inserted.

Types of Triggers:

1. **Row-level Trigger**: Executes once for each row affected. In the example, **incr_count** is a row-level trigger because of the presence of the **FOR EACH ROW** clause.
2. **Statement-level Trigger**: Executes once for the entire SQL statement, irrespective of the number of rows affected. In the example, **init_count** is a statement-level trigger since it lacks the **FOR EACH ROW** clause.

Further considerations:

- Triggers can be set to activate **before** or **after** a specified event (like **INSERT**, **UPDATE**, or **DELETE**).
- They can also be set to activate **instead of** the event itself.
- Execution can be deferred until the end of the transaction or asynchronously in a separate transaction.

In essence, triggers offer fine-grained control over database operations, allowing for a wide range of automated responses to specified events. However, it's crucial to use them judiciously, ensuring they don't lead to unwanted side-effects or complexities.

Triggers are potent tools that can address database changes, but they require careful management. The intertwined effects of multiple triggers can create intricate scenarios, making the upkeep of an active database challenging. In many instances, employing integrity constraints is a more straightforward alternative to using triggers.

The Complexity of Triggers

In an active database, whenever a modification statement is about to be executed, the DBMS checks for any activated triggers. If found, the DBMS evaluates the trigger's condition and executes its action if the condition holds. When multiple triggers are activated by a statement, the DBMS handles them all, but the sequence may not be predictable. Notably, a trigger's action might activate another trigger, even possibly reactivating itself - such triggers are deemed recursive. This chain reaction, combined with the unpredictable sequence of trigger processing, makes it hard to grasp the cumulative effect of multiple triggers.

Constraints vs. Triggers

Triggers are often employed to uphold database consistency. When this is the intent, it's crucial to assess if an integrity constraint (like a foreign key constraint) might serve the purpose better.

Unlike triggers, constraints are not defined operationally, making them simpler to understand. They offer the DBMS more optimization chances. While triggers respond to specific statements (INSERT, DELETE, or UPDATE), constraints guard against any inconsistency. This specificity makes constraints more transparent. But triggers offer flexibility in ensuring database integrity, as illustrated below:

- Consider an 'Orders' table with fields: iternid, quantity, customerid, and unitprice. When an order is placed, the first three fields are user-input. The fourth field's value, derived from an 'Items' table, is essential for recording complete order details. A trigger can automate this value's insertion, reducing manual inputs and eliminating potential inconsistencies.
- Expanding on this, when an order is made, additional checks might be needed, like verifying a purchase within a customer's credit limit. While a CHECK constraint could handle this, a trigger offers nuanced solutions, such as allowing slight over-limit purchases for long-standing customers or recommending credit limit hikes.

Diverse Roles of Triggers

Beyond ensuring integrity, triggers have varied applications:

- They can notify users about exceptional events mirrored in database updates. For instance, they might alert sales clerks about customer eligibility for extra discounts, driving more sales.
- Triggers can maintain logs for auditing or security reviews. For example, they might track a customer's order history, offering insights into potential credit limit adjustments.
- Triggers are useful for collecting statistics on table interactions. Some databases even leverage triggers for managing relation replicas. The applications of triggers span beyond these, encompassing aspects like workflow management and business rule enforcement.

3.7 Temporal databases

Temporal databases are specialized systems handling time-oriented data, essential in fields like medicine for patient histories or economics for tracking time-variant metrics like stock prices. They store data with relevant time periods, enabling queries about specific temporal states or historical changes. This facilitates understanding trends or events within a context, crucial in decision-making processes in various domains. Additionally, they maintain system-generated transaction times, allowing detailed change tracking. This dual temporal data nature expands query possibilities, driving research to enhance relational models for temporal data, impacting everyday activities reliant on time, like scheduling, medical diagnoses, or scientific observations.

A temporal database is specifically designed to manage time-variant data. Unlike traditional databases that record only the latest data, temporal databases maintain a historical record of data, which can be related to both real-world time (valid time) and system time

(transaction time). The concepts of time, especially in relation to databases and computing, involve several aspects like continuity, granularity, quanta, and structures like timelines, points, durations, and intervals.

- Continuous vs. Discrete Time:
  - Continuous: Philosophically, time can be seen as a constant flow, akin to a river, unbroken and seamless.
  - Discrete: For practicality in recording and analysis, time is often considered in distinct, equal segments, reflecting the ticking of a clock. Most computing systems, due to their digital nature, handle time discretely.
- Granularity:
  - This refers to the size of the smallest time unit considered in a scenario. Larger grains could be years or decades, suitable for geological scales, while smaller grains like seconds or milliseconds might apply to computing processes or microbiological observations. The chosen granularity depends on the context; for instance, analyzing high-frequency trading requires milliseconds, whereas studying climate change uses years.
- Time Quanta:
  - This concept is about the indivisible unit of time in a given context, often referred to as 'chronons' or 'time granules'. In computing and data modeling, the time quanta might be the smallest measurable interval the system can recognize or handle.
- Core Structures:
  - Points: Specific moments on a timeline, having no duration themselves. They're crucial for marking starts and ends of events.
  - Duration: Represents a quantity of time quanta, not tied to a specific start or end point. It's a measure of time length, such as "two weeks" or "50 milliseconds."
  - Interval: A segment of time defined by a starting point and an ending point. It encompasses all points from start to end, with an inherent assumption that the start occurs before the end.
  - Timeline: A graphical or conceptual representation of time, usually linear, used to illustrate time points, durations, and intervals in sequence. It helps visualize the order of events, their durations, and their relationships.

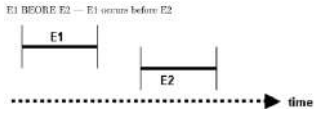
In temporal databases, these concepts govern how data is stored, queried, and analyzed. For instance, an interval might represent the valid period of a contract, with specific start and end points. Points might represent transaction times, and durations could represent lengths of events like machine uptimes or downtimes. The database's granularity and quanta will influence its precision and the nature of queries it can support. Understanding these concepts is vital for designing systems that effectively capture, store, and utilize time-based data.

The given descriptions illustrate the possible relationships or temporal patterns between two time intervals (or events) E1 and E2. Understanding these relationships is critical in various

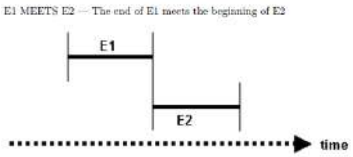
domains, such as scheduling, planning, or temporal databases, where events' relative timings can determine outcomes or logical flow.

To simplify and visualize, consider the following representations:

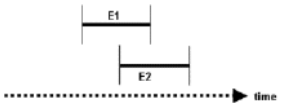
1. **E1 Precedes E2**:
   - E1 starts and ends before E2 begins.



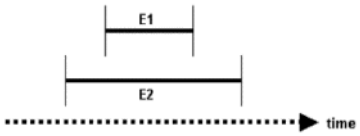
2. **E1 Meets E2**:
   - E1 ends at the same point in time that E2 begins.



3. **E1 Overlaps E2**:
   - E1 starts before E2 starts, but the end of E1 overlaps with the beginning of E2. E2 ends after E1 has ended.

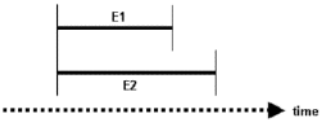


4. **E1 is During E2**:
   - E1 takes place entirely during the period that E2 exists, meaning E1 starts after E2 starts and E1 ends before E2 ends.



5. **E1 Follows E2**:
   - E1 starts after E2 has ended.

6. **E1 STARTS E2**
   - E1 starts at the same time that E2 starts (and E1 does not end after E2)



7. **E1 FINISHES E2**
   - E1 finishes at the same time that E2 ends (and E1 does not start before E2)

These temporal relationships can be crucial in systems like temporal databases, where the relative timing between events might influence queries or data interpretations. For instance, in a scheduling system, determining if one event overlaps another can prevent double-booking. Understanding and identifying these patterns can help in making informed decisions in various applications.

3.7.1 Types of Temporal Databases:

1. **Valid-time Temporal Databases:** These databases store information about when a certain fact is true in the real world. The valid time is typically provided by the users or the

applications interacting with the database, as it represents the actual time interval during which the fact holds in reality.

2. **Transaction-time Temporal Databases:** These databases store information about when a fact was stored or known to the system. It's essentially a record of the system's own knowledge, irrespective of when that fact was true in the real world.
3. **Bitemporal Databases:** As mentioned earlier, these databases store both valid time and transaction time. This allows them to answer queries like "What did we know and when did we know it?" as well as "When was this fact true in the real world?"

Advantages of Temporal Databases:

1. **Historical Analysis:** Since they store data changes over time, temporal databases allow for trend analysis, historical data querying, and auditing. This is especially beneficial for sectors like finance, healthcare, and research.
2. **Data Accuracy:** By storing data in its historical context, temporal databases can provide a more comprehensive and accurate view of data changes.
3. **Regulatory Compliance:** In some industries, keeping a record of historical data is not just beneficial, but also a regulatory requirement. Temporal databases make compliance easier.
4. **Audit Trail:** Provides a built-in mechanism to trace back changes, deletions, or updates, making it easier to identify discrepancies or errors.

Applications of Temporal Databases:

1. **Healthcare:** As mentioned, patient databases need to store historical data about patient health, treatments, and diagnoses.
2. **Finance:** Financial databases might need to store historical stock prices, account balances, and transaction details.
3. **Manufacturing and Industrial:** Sensor data, equipment maintenance logs, and production output metrics can be stored with timestamps for later analysis.
4. **Environmental Studies:** Tracking changes in weather patterns, animal migration, or pollution levels over time.
5. **Historical Research:** Storing events, data, and evidences with their valid time periods for academic and research purposes.

3.7.2 Interpreting Time in Relational Databases

Example of Temporal Database Query:

Imagine a database that keeps a record of employee salaries. A non-temporal query might ask, "What is John's salary?" In contrast, a temporal query might ask, "What was John's salary in 2018?" or "Show me the history of John's salary changes over the past five years."

In essence, temporal databases provide a time-aware approach to data management, making them indispensable in contexts where understanding the historical evolution of data is crucial.

Alright, let's look at a practical example of how a temporal database might be structured and how it can be queried. For simplicity, we'll take an example of an employee database that tracks salary changes over time.

Database Structure:

Table: Employee

EmployeeID	Name	Designation
001	John	Developer
002	Alice	Manager
003	Bob	Tester

Table: SalaryHistory

History ID	EmployeeID	Salary	Valid_From	Valid_To
1	001	$5000	2020-01-01	2020-12-31
2	001	$5500	2021-01-01	2021-12-31
3	002	$7000	2020-05-01	2020-12-31
4	002	$7500	2021-01-01	Present
5	003	$4000	2020-01-01	Present

In this database, the **Employee** table holds general details of the employees, and the **SalaryHistory** table keeps a record of salary changes over time.

Sample Queries:

1. **Find out John's current salary:**
   SELECT e.Name, s.Salary FROM Employee e JOIN SalaryHistory s ON e.EmployeeID = s.EmployeeID WHERE e.Name = 'John' AND s.Valid_To = 'Present';
2. **Get the history of Alice's salary changes:**
   SELECT e.Name, s.Salary, s.Valid_From, s.Valid_To FROM Employee e JOIN SalaryHistory s ON e.EmployeeID = s.EmployeeID WHERE e.Name = 'Alice';
3. **Identify employees who had a salary increase in 2021:**
   SELECT DISTINCT e.Name FROM Employee e JOIN SalaryHistory s ON e.EmployeeID = s.EmployeeID WHERE s.Valid_From = '2021-01-01';

This example provides a basic idea of how a temporal database can be used to track changes over time. More sophisticated systems may employ bitemporal data, advanced indexing, and more complex query constructs to cater to intricate temporal requirements.

Table: EmployeeContracts

EmployeeID	ContractType	StartDate	EndDate
101	Permanent	2020-01-01	NULL
102	Temporary	2019-05-01	2020-05-01
103	Temporary	2020-02-01	2021-02-01

In the above table, for EmployeeID 101, the contract is ongoing (hence the EndDate is NULL). For EmployeeID 102, the contract lasted from May 1, 2019, to May 1, 2020.

If a contract changes, instead of updating the record, a new record would be added with a new time span.

3.8 Self-Assessment questions and Exercises

Objective Question

1. Which of the following is NOT a benefit of using XML?
    a) Platform independence
    b) Self-descriptive
    c) Data encryption
    d) Extensible
2. Which tag in DTD denotes an element?
    a) <!ATTLIST>
    b) <!ELEMENT>
    c) <?XML>
    d) <!ENTITY>
3. Which is more expressive and flexible than DTD?
    a) XML Schema
    b) JSON Schema
    c) DTD
    d) XML-RPC
4. Which is a language used for querying XML data?
    a) XPath
    b) XSQL
    c) XLinq
    d) XQuery

5. Which protocol is commonly used to access web databases?
    a) SMTP
    b) HTTP
    c) FTP
    d) DHCP
6. ODBC is an API for:
    a) Sending emails
    b) Drawing graphics
    c) Accessing databases
    d) Web design
7. Which Java package contains JDBC classes?
    a) java.awt
    b) java.net
    c) java.util
    d) java.sql
8. Which PHP extension is used to connect to MySQL database?
    a) mysql_
    b) pdo_mysql
    c) mysqli
    d) mysqlnd
9. In the ECA model, what typically triggers an action?
    a) Event
    b) Condition
    c) Action
    d) None of the above
10. Which is a challenge in active databases?
    a) Too passive
    b) Handling infinite loops
    c) Low storage requirement
    d) Slow read speed
11. What special kind of information do temporal databases handle?
    a) Spatial data
    b) Music data
    c) Time-related data
    d) Video data
12. Which SQL data type specifically captures date and time?
    a) VARCHAR
    b) INT
    c) DATE
    d) FLOAT

The correct answers are: 1c, 2b, 3a, 4d, 5b, 6c, 7d, 8c, 9a, 10b, 11c, and 12c.

Short Questions

1. What is the primary purpose of the XML data model?
2. What role does a DTD play in XML documents, and how does it ensure structure consistency?
3. How does an XML Schema differ from a DTD, and why might one be preferred over the other?
4. Name one common language used for querying XML documents.
5. What is the primary advantage of web-based databases in modern web applications?
6. Why is ODBC important for the interoperability of database systems?
7. How does JDBC provide database connectivity in Java applications?
8. What is a common PHP extension used for MySQL database connectivity?
9. In the context of active databases, describe the role of the Event, Condition, and Action in the ECA model.
10. Name one challenge faced when implementing active databases as opposed to traditional databases.
11. How do temporal databases handle time-related data differently from standard relational databases?
12. What is the significance of timestamping in the context of relational databases?

Long Questions

1. Explain the structure and significance of the XML data model. How does it facilitate data interchange between heterogeneous systems? Illustrate with an example.
2. What is a DTD in the context of XML? How does it help in specifying the structure of an XML document? Differentiate between a DTD and an XML Schema.
3. Describe the advantages of XML Schema over DTD. How do data types and namespaces in XML Schema improve data representation and validation?
4. With the help of an example, demonstrate how one can query XML documents. What are the primary differences between querying XML and querying traditional relational databases?
5. Define web databases. How have they revolutionized data access and manipulation over the internet? Discuss their major benefits and challenges.
6. Elaborate on the role of ODBC in database connectivity. How does it provide a standardized interface for communication between databases and application programs?
7. Explain the JDBC architecture. How does it facilitate connection and interaction with databases using Java? Provide an example of connecting to a database and fetching data using JDBC.
8. Detail the methods and advantages of accessing a relational database using PHP. Illustrate with an example of connecting to a MySQL database, retrieving, and displaying records.

9. Describe the ECA model in the context of active databases. How do triggers based on the ECA model help in achieving database automation? Provide a practical use-case.
10. Discuss the primary challenges and considerations when designing and implementing an active database. How do triggers, rules, and event-driven actions impact performance and consistency?
11. Define temporal databases. How do they differ from traditional databases? Discuss their significance in storing time-variant data and provide an example scenario where temporal databases would be essential.
12. How is time interpreted and managed in relational databases? Discuss the techniques and structures that enable time-based queries and temporal data management.

13. Consider a relational database with the following tables:

Table: Employees

EmployeeID	FirstName	LastName	DepartmentID
1	John	Doe	101
2	Jane	Smith	102
3	Mike	Johnson	101

Table: Departments

DepartmentID	DepartmentName
101	IT
102	HR

a) Convert the above relational model to an XML representation.

b) Using the XML representation from "a", create an XML DTD that enforces the following rules
   - Each employee must have a unique EmployeeID.
   - The FirstName and LastName elements are required.
   - The DepartmentID must be a reference to a valid DepartmentID in the Departments table.
   - Each department must have a unique DepartmentID.
   - The DepartmentName is required.

c) Extend the XML DTD from "b" to a full XML Schema. Add the following constraints:
   - The Department ID in the Employees table must be unique.
   - Each employee must belong to a department (no null values allowed for Department ID).
   - The Department ID in the Departments table must be unique.
   - The Department Name is limited to a maximum length of 50 characters.

3.9 Summary

XML provides a standardized format for structuring data, with the XML Data Model serving as its foundational structure. Data Type Definitions (DTD) offer a means to define the structure and data types in XML documents, whereas XML Schema provides a more extensive and flexible way to describe the content of an XML document. For querying XML data, XML Querying techniques are employed. Web databases can connect to applications through methodologies like Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). PHP, a server-side scripting language, can access relational databases, enhancing web application functionality. Active databases rely on the Event Condition Action Model, triggering predefined actions when certain conditions are met, though their design and implementation can be challenging. Additionally, temporal databases capture data related to time, necessitating specific strategies to interpret time in relational database contexts.

Keywords

XML, DTD, XML Schema, ODBC, JDBC, ECA model, Temporal Database

3.10 Further readings

1. Beginning XML" by Joe Fawcett, Danny Ayers, and Liam R. E. Quin
2. XML for the World Wide Web" by Elizabeth Castro
3. Web Database Applications with PHP & MySQL" by Hugh E. Williams and David Lane
4. Snodgrass, Richard T., ed. "The TSQL2 temporal query language." Springer Science & Business Media, 1995.
5. Database System Concepts by Henry F. Korth, S. Sudarshan, Abraham Silberschatz , Seventh Edition, March 2019.

UNIT 4

NoSQL DATABASES

CONTENTS

4.1 Introduction

NoSQL and SQL databases represent two different paradigms of data storage and retrieval. Traditional SQL databases, like MySQL, Oracle, and PostgreSQL, store data in structured tables using a fixed schema and rely on the Structured Query Language (SQL) for data manipulation. They are known for their ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring reliable transactions.

In contrast, NoSQL databases, such as MongoDB, Cassandra, and Couch base, offer more flexible storage mechanisms like document, key-value, column-family, and graph formats. They often don't require a fixed schema, allowing them to scale more easily and adapt to varied and rapidly changing datasets. The CAP theorem posits that it's impossible for distributed data systems to simultaneously achieve Consistency, Availability, and Partition tolerance. This theorem plays a pivotal role in understanding trade-offs in NoSQL systems. As businesses evolve, some find it beneficial to migrate from traditional RDBMS to NoSQL to handle large volumes of unstructured data, enhance scalability, or reduce latency. MongoDB, a popular NoSQL database, supports CRUD (Create, Read, Update, Delete) operations and offers features like sharding for horizontal scaling and replication for data availability and fault tolerance. Modern web application development has leveraged MongoDB's capabilities, often integrating it with programming languages like PHP and Java to create dynamic, scalable, and efficient applications.

4.2 Learning Objectives

- Understand the fundamental differences between NoSQL and SQL databases.
- Analyze the trade-offs involved in optimizing for any two of the three CAP principles.
- Identify scenarios where migration from RDBMS to NoSQL is advantageous.
- Understand the importance of indexing in MongoDB for efficient querying
- Define sharding and its significance in MongoDB.
- Understand the roles of primary and secondary nodes in replication.
- Utilize appropriate drivers and libraries for MongoDB connectivity in both PHP and Java.

4.3 Overview

SQL, short for 'Structured Query Language,' has served as a foundational programming language for managing data in relational database management systems (RDBMS) since the 1970s. In its early years, SQL databases focused on minimizing data duplication due to the high cost of storage. In today's context, SQL remains a widely adopted tool for querying relational databases. These databases store data in structured tables, where records are linked in various ways. A single record in one table can establish links to one or many records in other tables, and vice versa. These relational databases excel in efficient data storage and retrieval, enabling them to handle vast amounts of data and complex SQL queries effectively.

NoSQL databases emerged as an alternative to traditional relational (SQL) databases, designed to handle vast amounts of data, offer scalability, and accommodate the more flexible, varied data models that modern applications often require. While SQL databases use structured query language (SQL) for defining and manipulating data, NoSQL databases can store unstructured or semi-structured data and use a variety of query languages. The CAP theorem is fundamental when discussing databases. It posits that a distributed system can achieve at most two out of three guarantees: Consistency, Availability, and Partition tolerance. While many RDBMS focus on consistency and availability, NoSQL systems often emphasize partition tolerance, making them suitable for distributed environments.

MongoDB is a popular NoSQL database, known for its document-oriented model. It stores data in BSON format, a binary representation of JSON. Basic CRUD operations (Create, Read, Update, Delete) are intuitive in MongoDB, leveraging JSON-like documents. One of MongoDB's powerful features is "sharding," which distributes data across multiple servers, enhancing horizontal scalability. Moreover, its replication ensures data availability and redundancy. In the realm of web application development, MongoDB's flexibility and scalability make it a preferred choice when paired with popular languages like PHP and Java. This pairing empowers developers to efficiently build robust, data-intensive applications.

4.4 NoSQL Database vs. SQL Databases

NoSQL (often interpreted as "not only SQL") is a term used to describe a set of databases that provide mechanisms for data storage and retrieval that differ from traditional relational databases managed by SQL (Structured Query Language). Here's a brief overview:

Characteristics of NoSQL databases:

1. **Scalability:** They are particularly designed for large-scale data storage and operation, allowing for horizontal scalability.
2. **Flexibility:** NoSQL databases can manage unstructured and semi-structured data, offering more flexibility in terms of data models.
3. **Performance:** They can provide faster data operations as they can be optimized for specific data models (like key-value or document).

Types of NoSQL Databases:

1. **Document-oriented:** Stores data in documents (often JSON format). Example: MongoDB.
2. **Key-Value Stores:** Data is stored as a collection of key-value pairs. Examples: Redis, Riak.
3. **Column-oriented:** Data is stored in columns instead of rows. Examples: Cassandra, HBase.
4. **Graph databases:** Designed for data whose relations are well represented as a graph. Examples: Neo4j, OrientDB.

Advantages:

- **Schema-less:** Unlike relational databases, you don't need a fixed schema in many NoSQL databases. This makes them adaptive to changing data models.
- **Distributed Computing:** They are designed for distributed data, allowing for efficient scaling out.

Drawbacks:

- **Maturity:** Relational databases have been around for decades, making them more mature and stable in many scenarios.
- **Standardization:** SQL offers a standard language that many professionals are familiar with, while NoSQL databases often lack such standardization.

Use Cases:

- **Big Data and Real-Time Applications:** NoSQL databases are built to handle vast amounts of data, making them apt for big data applications.
- **Content Management and Catalogs:** The flexible schema can easily adapt to changing data.
- **Mobile and User Data Stores:** They can accommodate the varied, dynamic user-generated data typical in mobile applications.

In a relational database system (RDBMS), adding capacity usually involves investing in expensive hardware upgrades, such as faster CPUs, additional RAM, and more advanced networking components. This is known as vertical scaling or scaling up. It essentially means increasing the power of existing hardware resources to handle larger workloads. Conversely, NoSQL databases prioritize low latency and high resilience and are designed to operate on distributed clusters of nodes. This strategy is referred to as horizontal scaling or scaling out. To expand the capacity of a NoSQL database, administrators simply add more nodes, a relatively straightforward process in modern cloud environments. NoSQL databases are inherently engineered for elastic capacity, enabling organizations to adjust their data infrastructure to meet business demands, even during periods of heightened demand or when addressing node failures and network issues as depicted in Figure 4.1

However, horizontal scaling in NoSQL databases comes with its own set of trade-offs. While adding commodity hardware to a cluster can be cost-effective in terms of software licenses and subscriptions, managing larger clusters with more nodes can lead to increased operationaloverhead and administrative costs. Larger clusters generate more alerts and require

more attention.

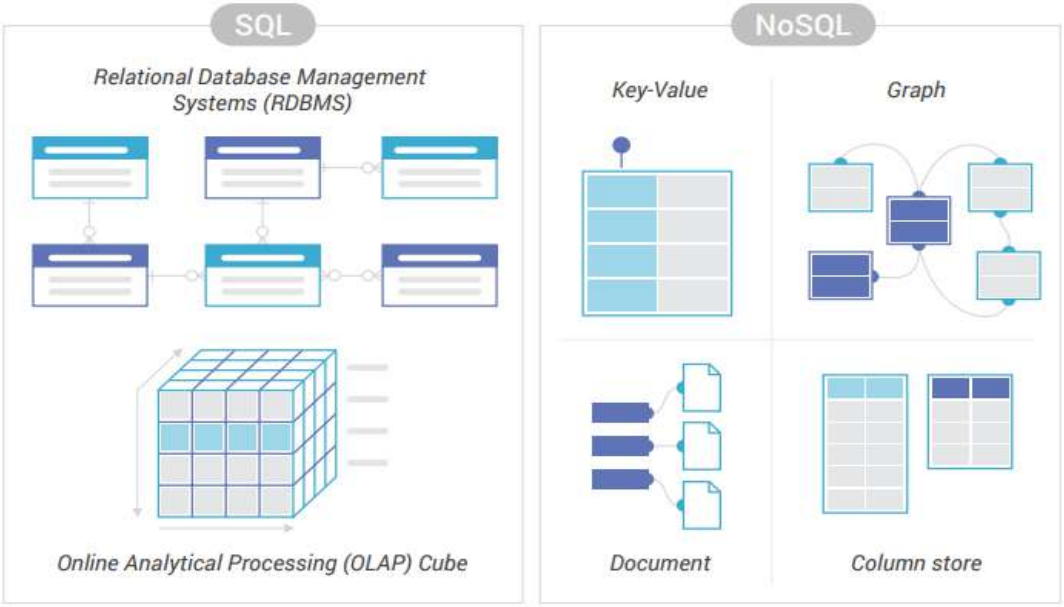


Figure 4.1 SQL Vs NoSQL

NoSQL databases have grown in popularity alongside the rise of web and cloud applications, especially when scalability and quick development are crucial. However, the choice between SQL and NoSQL should be based on the specific needs of the project.

Attribute	SQL(RDBMS)	NoSQL
Data Model	Structured data with tables, rows, and columns.	Structured, semi-structured, and unstructured data.
Schema	Requires predefined schema.	Typically schema-less.
Query Language	SQL (Structured Query Language).	Varies (e.g., MongoDB uses BSON).
Scalability	Scales vertically Scale up – bigger load , bigger server	Scales horizontally Scale out- distribute data acroos multiple hosts seamlessly
Transaction Properties	ACID properties (Atomicity, Consistency, Isolation, Durability).	Generally, BASE properties (Basically Available, Soft state, Eventually consistent).
Economics	Expensive proprietary servers to manage data	Cluster of cheap commodity server to manage the data and transaction volumes
DBA (Data Base Administrator) Specialists	Require highly trained expert to monitor DB	Require less management, automatic repair and simple data models
Examples	MySQL, PostgreSQL, Oracle, SQL Server.	MongoDB, Cassandra, Redis, Neo4j.
Ideal For	Complex queries, stable structured data.	Rapidly changing data, large datasets, high write loads.

In NoSQL databases, transactions are handled with a focus on **BASE** (**B**asically **A**vailable, **S**oft State, **E**ventual Consistency) properties rather than the strict ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) properties commonly associated with traditional relational databases, RDB (ACID) and NoSQL (BASE) relation is shown in Figure 4.2. Here's how transactions are typically managed in NoSQL databases with an emphasis on BASE:

Basically Available: NoSQL databases prioritize high availability and are designed to remain operational, even when network partitions or other failures occur. During a transaction, the system aims to remain accessible to users and applications, ensuring that read and write operations can be performed.

Soft State: NoSQL databases acknowledge that data may be in a soft or temporarily inconsistent state during transactions. Updates and changes may not be immediately synchronized across all

nodes in the system, allowing for flexibility and responsiveness. This can result in temporary data inconsistency.

Eventual Consistency: Eventual consistency is a fundamental aspect of NoSQL databases. It means that, over time, all replicas of the data in the distributed system will converge to the same consistent state. This convergence may not occur immediately but eventually ensures that all nodes have the same data.

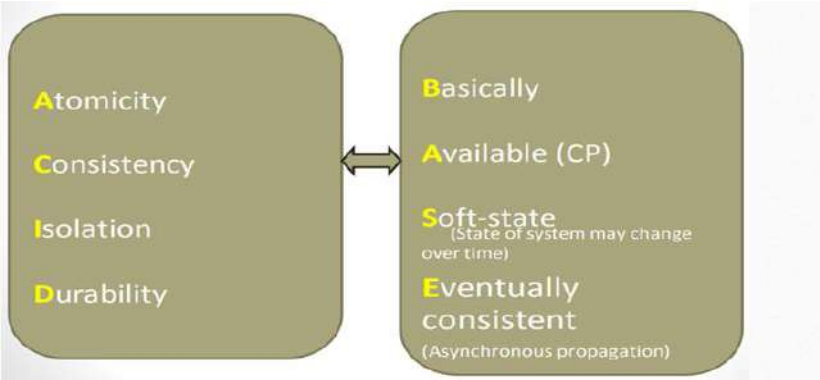


Figure 4.2 RDB ACID to NoSQL BASE

To provide these BASE properties while handling transactions, NoSQL databases often use techniques such as:

Asynchronous Replication: Changes made to data within a transaction might not be immediately propagated to all nodes. Instead, these changes are replicated asynchronously, potentially causing temporary data inconsistencies until all nodes synchronize.

Conflict Resolution: NoSQL databases may employ conflict resolution mechanisms to resolve conflicts that arise when different nodes have different versions of the same data.

Versioning: Some NoSQL databases use versioning to track changes to data, allowing for the identification and resolution of conflicts in a distributed environment.

It's important to understand that the level of support for transactions and the specific implementation of BASE properties can vary among different NoSQL databases. When working with NoSQL databases, you should carefully consider the database system's documentation and capabilities to ensure that it aligns with your application's requirements for data consistency and availability.

4.5 CAP Theorem

The CAP Theorem, often referred to as Brewer's Theorem, is a fundamental principle applied to distributed data systems. It was formulated by Eric Brewer in 2000. The theorem states that, in the presence of a network partition, a distributed system can guarantee only two out of the following three properties:

1. **Consistency (C)**: All nodes in the system see the same data at the same time. In other words, every read receives the most recent write.
2. **Availability (A)**: Every request (read or write) made to the system receives a response, without a guarantee that it contains the most recent version of the data.
3. **Partition tolerance (P)**: The system continues to function even when network partitions occur; meaning that communication between nodes is lost or delayed.

According to the CAP theorem (Figure 4.3), a distributed data store can only guarantee two out of the three properties at any given time.

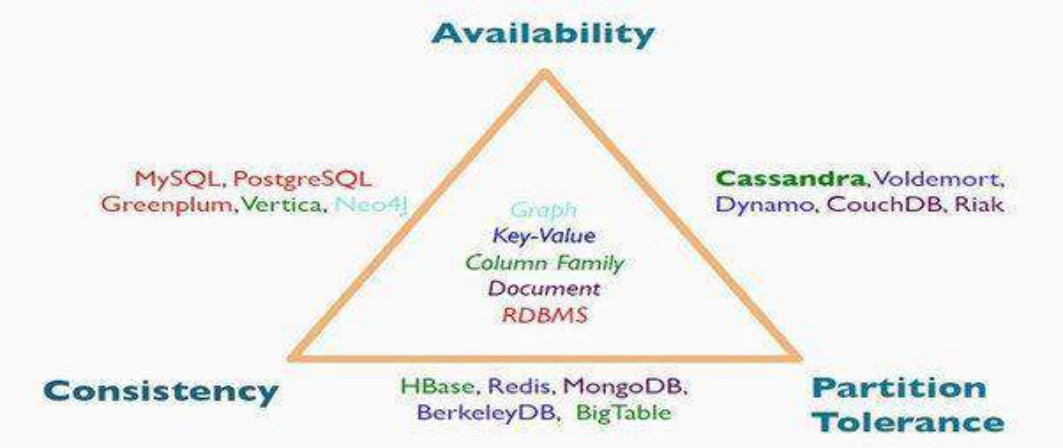


Figure 4.3 CAP Theorem

Implications:

1. **CP (Consistency and Partition tolerance)**: CP gives precedence to data consistency over availability. During a network partition, the system might become unavailable. Systems such as HBase and BigTable adhere to CP. For instance, in scenarios like constructing a distributed file system, emphasizing consistency and partition tolerance implies that during network partitions, the system will temporarily become unavailable to maintain data consistency.
2. **CA (Consistency and Availability)**: CA prioritizes consistency and availability but cannot handle network partitions. In real-world distributed systems, partition tolerance is often essential, making CA a less likely combination. In some scenarios, prioritizing consistency and availability over partition tolerance is necessary. For example, a traditional relational database might prioritize data consistency and high availability in a single data center, even if it entails the system being unable to handle network partitions.
3. **AP (Availability and Partition tolerance)**: AP prioritizes availability over consistency. During a network partition, reading and writing data remain possible, but immediate consistency across all nodes may not be guaranteed. ventually, the data becomes consistent (eventual consistency). Examples of systems that follow AP are Cassandra, Couchbase,

and DynamoDB. When building web applications or content delivery systems, prioritizing availability and partition tolerance means that the system remains responsive, even in the presence of network partitions, but it may not provide the most up-to-date data in some cases.

It's important to note that the specific trade-offs between consistency, availability, and partition tolerance can be tailored according to the specific needs of a system. There is a spectrum of possibilities between these three extremes. Numerous distributed databases and systems strive to strike a balance among these properties, aligning with their respective use cases and requirements. The CAP theorem helps in understanding the inherent trade-offs in distributed systems and guides the design and selection of databases based on use cases.

4.6 Migrating from RDBMS to NOSQL

Migrating from a traditional Relational Database Management System (RDBMS) to a NoSQL database involves a series of steps and considerations due to the fundamental differences between the two systems. Here's an overview:

1. **Assessing the Need:**
   - **Scale**: NoSQL databases can handle massive amounts of data and high-velocity read-write operations, making them suitable for large-scale applications.
   - **Flexibility**: If application requires a flexible schema that can evolve over time, NoSQL can be a good fit.
   - **Data Model**: Understand if your data is better represented in key-value, document, column-family, or graph format.

2. **Choosing the Right NoSQL Database:**

   Different NoSQL databases serve different purposes:

   - **Document Stores** (e.g., MongoDB): Good for content management or catalogs.
   - **Key-Value Stores** (e.g., Redis): Suitable for caching and session management.
   - **Column-family Stores** (e.g., Cassandra): Ideal for analytics and real-time big data use cases.
   - **Graph Databases** (e.g., Neo4j): Used for connected data, like social networks.

3. **Data Modeling:**
   Unlike RDBMS where you design around a fixed schema and relations, in NoSQL, you design based on your query requirements. The denormalization of data might be necessary, and you might need to store data redundantly to optimize for read operations.

4. **Migration:**
   - **ETL Process**: Use Extract, Transform, Load (ETL) tools to move data from RDBMS to NoSQL.
   - **Consistency Check**: Ensure that the data in the NoSQL database matches the original data in the RDBMS.

5. **Update Application Logic:**
   Adjust your application's data access layer to interact with the NoSQL database. This might involve using new drivers, rewriting CRUD operations, and updating query mechanisms.

6. **Testing:**
   Thoroughly test the application to ensure that it works as expected with the NoSQL backend. This includes functionality, performance, and scalability testing.

Challenges:

- **Consistency**: NoSQL databases often provide eventual consistency, which might not be suitable for all use cases.
- **Skill Set**: Your team might need training, as working with NoSQL databases requires a different set of skills compared to RDBMS.
- **Complexity**: Some operations, like JOINs, which are straightforward in RDBMS, might require more intricate solutions in NoSQL.

To summarize, although NoSQL databases provide numerous advantages in scalability and flexibility, migrating from RDBMS demands meticulous planning, selecting the appropriate database, and ensuring the application continues to fulfill user and business requirements.

4.7 MongoDB

MongoDB was created by Dwight Merriman and Eliot Horowitz, who encountered development and scalability issues with traditional relational database approaches while building web applications at DoubleClick, an online advertising company that is now owned by Google Inc. The name of the database was derived from the word 'humongous' to reflect its capacity to handle large amounts of data. Merriman and Horowitz co-founded 10Gen Inc. in 2007 to commercialize MongoDB and related software. The company was later renamed MongoDB Inc. in 2013 and went public in October 2017, trading under the ticker symbol MDB.

The database management system (DBMS) was released as open-source software in 2009 and has been continuously updated since. Various organizations, such as the insurance company MetLife, have employed MongoDB for customer service applications. Other websites, including Craigslist, have utilized it for data archiving. The CERN physics lab has harnessed MongoDB for data aggregation and discovery, while The New York Times has leveraged it to support a form-building application for photo submissions.

MongoDB is an open-source NoSQL database management program. NoSQL (Not only SQL) serves as an alternative to traditional relational databases and proves highly effective for handling large sets of distributed data. MongoDB is a versatile tool for managing document-oriented information and facilitating the storage and retrieval of data. MongoDB is widely used for high-

volume data storage, enabling organizations to store vast amounts of data while maintaining high performance. Additionally, organizations leverage MongoDB for its capabilities in ad-hoc querying, indexing, load balancing, aggregation, server-side JavaScript execution, and other advanced features.

Structured Query Language (SQL) is a standardized programming language employed for managing relational databases. SQL organizes data into schemas and tables, where each table adheres to a fixed structure. In contrast to relational databases, which use tables and rows, MongoDB's architecture revolves around collections and documents. Documents consist of key-value pairs, serving as MongoDB's fundamental data unit. Collections, equivalent to SQL tables, house sets of documents as depicted in Figure 4.4. MongoDB offers support for a wide range of programming languages, including C, C++, C#, Go, Java, Python, Ruby, and Swift.

4.7.1 How does MongoDB work?

MongoDB environments offer users a server to create databases using MongoDB, where data is stored as records comprised of collections and documents. These documents contain the data users wish to store in the MongoDB database, represented as field and value pairs, serving as the fundamental data unit. They resemble JavaScript Object Notation (JSON) but use a variant called Binary JSON (BSON), advantageous for its capacity to handle more data types. The fields in these documents are analogous to the columns in a relational database. They can hold various data types, including other documents, arrays, and arrays of documents, as per the MongoDB user manual. Moreover, each document includes a primary key serving as a unique identifier. The structure of a document remains flexible, allowing for modifications by adding or removing new or existing fields.

Collections, functioning as the counterparts of relational database tables, comprise sets of documents and can accommodate various types of data. It's crucial to note that data within a collection cannot span across different databases. MongoDB users have the flexibility to create multiple databases, each containing multiple collections. The mongo shell represents a standard component of open-source MongoDB distributions. Once MongoDB is installed, users can connect the mongo shell to their running MongoDB instances. Serving as an interactive JavaScript interface to MongoDB, the mongo shell empowers users to query, update data, and execute administrative operations.

The BSON document storage and data interchange format provides a binary representation of JSON-like documents. Another key feature is automatic sharding, enabling the distribution of data from a MongoDB collection across multiple systems to achieve horizontal scalability in line with escalating data volumes and throughput demands. The NoSQL DBMS employs a single-master architecture for data consistency, accompanied by secondary databases that maintain copies of the

primary database. Operations are automatically replicated to these secondary databases to ensure automatic failover.

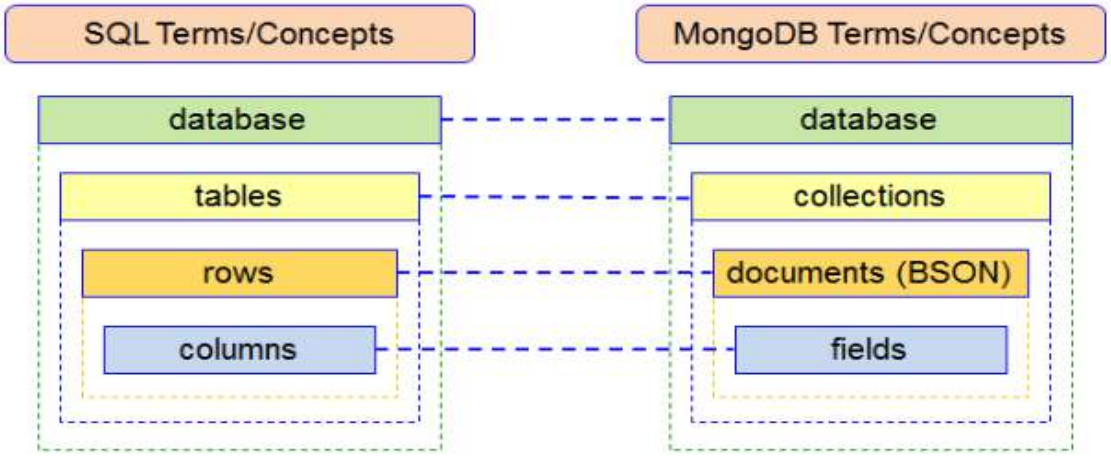


Figure 4.4 SQL vs MongoDB

An organization may choose to implement MongoDB for the following purposes:

- Storage: MongoDB is a versatile database capable of storing large volumes of both structured and unstructured data. It offers vertical and horizontal scalability, and indexing is employed to enhance search performance. MongoDB supports a variety of search methods, including field, range, and expression queries.
- Data Integration: MongoDB facilitates data integration for applications, making it suitable for use in hybrid and multi-cloud environments.
- Complex Data Structures: MongoDB's document-oriented nature allows for the embedding of documents to describe nested data structures. This flexibility enables the database to handle variations in data effectively.
- Load Balancing: MongoDB is adept at distributing workloads across multiple servers, promoting efficient load balancing for high-performance applications.

4.7.2 Features of MongoDB

MongoDB offers several key features, including:

- Replication: MongoDB uses replica sets consisting of primary and secondary servers to ensure high availability. The primary server handles read and write operations, while the secondary maintains a copy of the data. If the primary fails, the secondary takes over.
- Scalability: MongoDB supports both vertical and horizontal scaling. Vertical scaling involves increasing the power of existing machines, while horizontal scaling involves adding more machines to expand resources.

- Load Balancing: MongoDB efficiently manages load balancing without requiring a separate, dedicated load balancer. This can be achieved through vertical or horizontal scaling.
- Schema-less: MongoDB is a schema-less database, allowing it to handle data without a predefined schema or blueprint. This flexibility is particularly useful for evolving data structures.
- Document-Oriented: MongoDB stores data in documents, using key-value pairs instead of traditional rows and columns. This document-based structure offers greater flexibility when compared to SQL databases.

4.7.3 How to get started with MongoDB

MongoDB runs on most platforms and supports both 32-bit and 64-bit architectures. MongoDB is available as a binary or a package. In production environments, it is recommended to use 64-bit MongoDB binaries.

Installation:

Download MongoDB: Visit the official MongoDB website and download the community server version suitable for your operating system.

Installation: Follow the installation instructions provided for your specific OS. MongoDB provides straightforward installation processes for Windows, macOS, and Linux.

MongoDB Setup:

Start MongoDB: Once installed, start the MongoDB server. For most systems, you can use a command like mongod/ mongo to start the server.

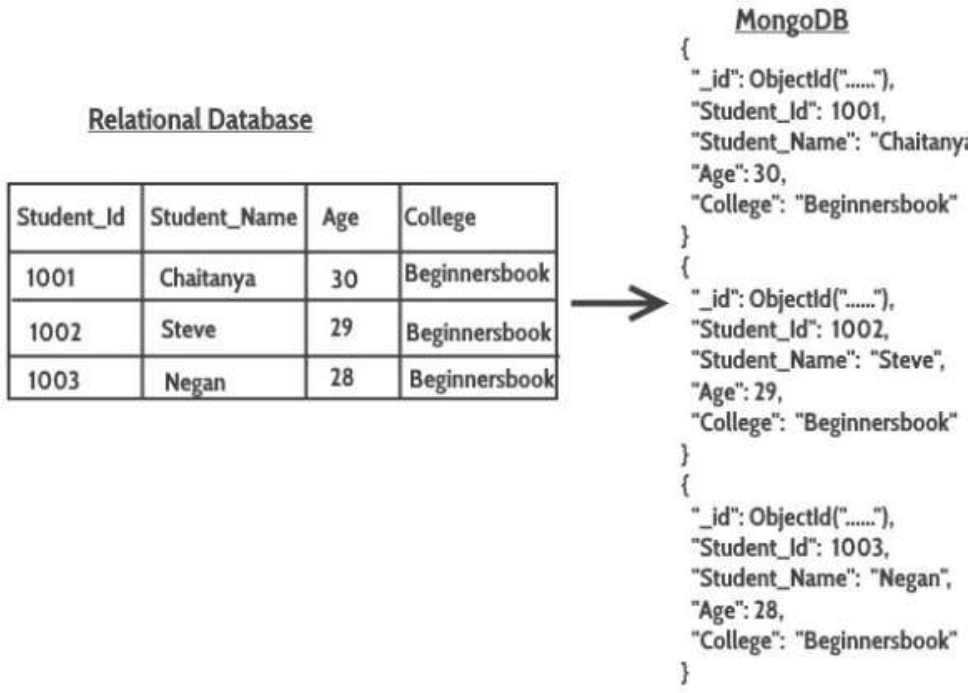
Access MongoDB: Use the MongoDB shell to interact with the database. Run mongo in your command line or terminal.

- mongod stands for “Mongo Daemon”. mongod is a background process used by MongoDB. The main purpose of mongod is to manage all the MongoDB server tasks. For instance, accepting requests, responding to client, and memory management.

- mongo is a command line shell that can interact with the client (for example, system administrators and developers).

BASIC COMMANDS	
use my Database	Creates or switches to 'myDatabase'
db.createCollection("myCollection")	Creates a collection called 'myCollection' in the current database
db.myCollection.insertOne({ name: "John", age: 30 })	Inserts a document into 'myCollection'
db.myCollection.find()	Retrieves all documents in 'myCollection'
db.myCollection.updateOne({ name: "John" }, { $set: { age: 31 } })	Updates the age of the document with name 'John'
db.myCollection.deleteOne({ name: "John" })	Deletes a document from 'myCollection'

In SQL databases, these are similar to Tables.



4.7.4 CRUD Operations

In MongoDB, CRUD operations refer to the fundamental operations for managing data in a database. "CRUD" stands for Create, Read, Update, and Delete, and these operations are essential for working with data in a MongoDB database.

CRUD operations create, read, update, and delete documents.

Create Operations
Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

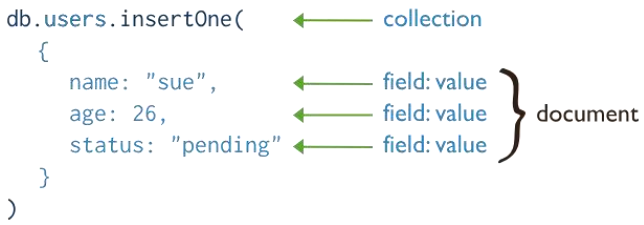
db.collection.insert(<document>)
db.collection.save(<document>)
db.collection.update(<query>, <update>, { upsert: true })

MongoDB provides the following methods to insert documents into a collection:

db.collection.insertOne()

db.collection.insertMany()

In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.



- db.users.insertOne( ): This is a command to insert a document into the "users" collection in the database. The arrow pointing to it labels it as "collection", which indicates that "users" is the collection where the document will be inserted.

- { name: "sue", age: 26, status: "pending" }: This is a document being inserted into the collection. A document in MongoDB is a set of key-value pairs. The arrow pointing to it labels it as "document".

- name: "sue", age: 26, and status: "pending": These are the individual fields (key-value pairs) within the document. Each of them has an arrow pointing to them with the label "field:value", indicating the structure of each field in the document.

In MongoDB, data is stored in collections as documents. Documents are essentially sets of key-value pairs, with each pair representing a field in the document. The image is demonstrating this structure by showing how to insert a single document into a collection and how the different parts of the command relate to the components of MongoDB's data model.

Logical Operators

MongoDB provides logical operators. The picture below summarizes the different types of logical operators.

Operand	Example	Meaning
&&	$variable1 && $variable2	Are both values true?
||	$variable1 || $variable2	Is at least one value true?
AND	$variable1 AND $variable2	Are both values true?
XOR	$variable1 XOR $variable2	Is at least one value true, but NOT both?
OR	$variable1 OR $variable2	Is at least one value true?
!	!$variable1	Is NOT something

Read Operations

Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

- db.collection.find(<query>, <projection>)
- db.collection.findOne(<query>, <projection>)
- db.collection.find()

Specify query filters or criteria that identify the documents to return.

```
db.users.find(                          ← collection
   { age: { $gt: 18 } },                ← query criteria
   { name: 1, address: 1 }              ← projection
).limit(5)                              ← cursor modifier
```

Query Operators

Name	Description
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a loginal AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

- db.users.find( ): This command is utilized to retrieve documents from the "users" collection in the database. The arrow pointing to it labels it as "collection", indicating that "users" is the collection being queried.

- { age: { $gt: 18 } }: This is the query criteria specifying the condition for the search. The arrow pointing to it labels it as "query criteria". This criterion is specifically looking for documents where the "age" field has a value greater than 18.

- { name: 1, address: 1 }: This part of the query is called projection. It specifies which fields to include in the results. The number "1" indicates the field should be included, while a "0" would indicate exclusion. The arrow pointing to this portion labels it as "projection".

- limit(5): This is a cursor modifier. It limits the result set to 5 documents. The arrow pointing to this method labels it as "cursor modifier".

In essence, the MongoDB command presented is querying the "users" collection for documents where the "age" is greater than 18. It will return only the "name" and "address" fields for up to 5 matching documents. The image elucidates the structure of the command by demarcating the function and purpose of each segment

Update Operations

Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

db.collection.update(<query>, <update>, <options>)

db.collection.updateOne()

db.collection.updateMany()

db.collection.replaceOne()

In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

Delete Operations

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection

db.collection.remove(<query>, <justOne>)

db.collection.deleteOne()

db.collection.deleteMany()

In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.

- db.users.updateMany(): This command is employed to update multiple documents in the "users" collection in the database. The arrow pointing to this command labels it as "collection", highlighting that "users" is the collection being modified.

- { age: { $lt: 18 } }: This portion represents the filter for which documents should be updated. It's identified by the arrow pointing to it with the label "update filter". According to this criterion, the command will target documents where the "age" field has a value less than 18.

- { $set: { status: "reject" } }: This part represents the update action. The arrow pointing to this portion labels it as "update action". It indicates that for all the documents that match the previous criteria (age less than 18), the "status" field should be set or updated to the value "reject".

In summary, the MongoDB command depicted is updating the "users" collection, setting the "status" field to "reject" for all documents where the "age" is less than 18.

- db.users.deleteMany(): This command is used to delete multiple documents from the "users" collection in the database. The arrow pointing to this command identifies it as "collection", indicating that the operation is being performed on the "users" collection.

- { status: 'reject' }: This section represents the criteria or filter for determining which documents should be deleted. The arrow pointing to this segment labels it as "delete filter". As per this filter, the command will target documents where the "status" field has the value "reject".

In essence, the depicted MongoDB command will remove all documents from the "users" collection that have a "status" field set to "reject".

4.7.5 MongoDB sharding

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high-throughput operations. Database systems with large data sets or high-throughput applications can challenge the capacity of a single server. For example, high query rates can exhaust the CPU capacity of the server, while working set sizes larger than the system's RAM stress the I/O capacity of disk drives. There are two methods for addressing system growth**:**

vertical and horizontal scaling

- Vertical Scaling involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space. Limitations in available technology may restrict a single machine from being sufficiently powerful for a given workload. Additionally, cloud-based providers often have hard ceilings based on available hardware configurations. As a result, there is a practical maximum for vertical scaling.

- Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required. While the overall speed or capacity of a single machine may not be high, each machine handles a subset of the overall workload, potentially providing better efficiency than a single high-speed, high-capacity server. Expanding the capacity of the deployment only requires adding additional servers as needed, which can be a lower overall cost than investing in high-end hardware for a single machine. The trade-off is increased complexity in infrastructure and maintenance for the deployment. MongoDB supports horizontal scaling through sharding.

A MongoDB sharded cluster consists of the following key components as depicted in Figure 4.5.

- **Shard:** Each shard contains a portion of the sharded data. For high availability and data redundancy, shards can be deployed as replica sets.
- **Mongos:** The mongos component acts as a query router, providing an interface between client applications and the sharded cluster. Additionally, mongos support hedged reads to reduce latencies.
- **Config Servers:** These servers store crucial metadata and configuration settings essential for the cluster's functioning.

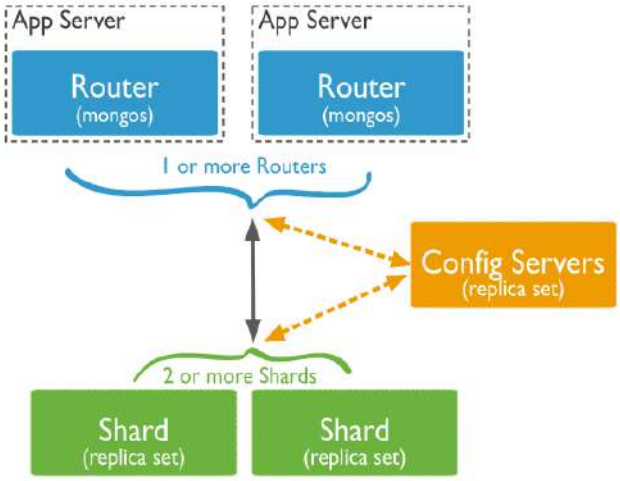


Figure 4.5 Key Components of MongoDB sharded cluster

In MongoDB, data sharding operates at the collection level. This means that each collection within the database can be individually sharded, allowing its data to be distributed across available shards in the cluster. Sharding collections instead of entire databases provides enhanced flexibility and granularity in managing data distribution. This approach facilitates the sharding of specific collections that benefit most from horizontal scaling, while leaving others unsharded if they do not require the same level of scaling.

MongoDB utilizes a shard key to distribute documents of a collection among shards, comprising one or more fields within the documents. Documents in sharded collections might not contain shard key fields. In such cases, the missing fields are treated as having null values during distribution across shards, although they are not considered null when routing queries

To shard a populated collection, an index beginning with the shard key is required. When sharding an empty collection and the necessary index for the specified shard key is absent, MongoDB will automatically create it. The choice of shard key profoundly affects the performance, efficiency, and scalability of a sharded cluster. Even a well-equipped cluster with top-notch hardware can face bottlenecks due to an inadequately chosen shard key. Additionally, the shard key's selection, alongside its supportive index, can dictate the sharding strategy your cluster adopts.

Advantages of Sharding

- **Reads/Writes**

MongoDB distributes the read and write workloads across the shards in the sharded cluster, enabling each shard to process a subset of cluster operations. By adding more shards, both read and write workloads can be horizontally scaled across the cluster.

For queries that include the shard key or the prefix of a compound shard key, mongos can target the query at a specific shard or set of shards. These targeted operations are generally more efficient than broadcasting to every shard in the cluster.

- **Storage Capacity**

Sharding distributes data across the shards in the cluster, allowing each shard to store a subset of the total cluster data. As the dataset grows, adding additional shards increases the storage capacity of the cluster.

- **High Availability**

The deployment of config servers and shards as replica sets provides increased availability. Even if one or more shard replica sets become completely unavailable, the sharded cluster can continue to perform partial reads and writes. In other words, while data on the unavailable shard(s) cannot be accessed, reads or writes directed at the available shards can still succeed.

Considerations Before Sharding

The complexity and requirements of a sharded cluster infrastructure necessitate meticulous planning, execution, and maintenance. Once a collection is sharded, MongoDB does not offer a method to unshard it. While it is possible to reshard your collection later on, it is crucial to carefully consider your shard key choice to avoid scalability and performance issues. For a deeper understanding of the operational requirements and limitations of sharding your collection. If queries do not include the shard key or the prefix of a compound shard key, the mongos performs a broadcast operation, querying every shard in the cluster. Such scatter/gather queries can be time-consuming..

Sharded and Non-Sharded Collections

A database can contain both sharded and unsharded collections. Sharded collections are partitioned and distributed across the shards in the cluster, while unsharded collections reside on a primary shard. Each database has a designated primary shard. The Figure 4.6 shows the collection 1 as a sharded collection and the collection 2 as an unsharded collection.

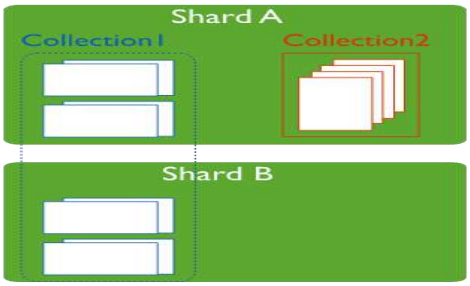


Figure 4.6 **Sharded (Collection 1) and Non-Sharded (Collection 2)**

Connecting to a Sharded Cluster

To interact with any collection in the sharded cluster, a connection to a mongos router is required. This includes both sharded and unsharded collections. Clients should never connect directly to a single shard to perform read or write operations. Connection to a mongos can be established in the same manner as to a mongod, either using mongosh or a MongoDB driver as given in Figure 4.7

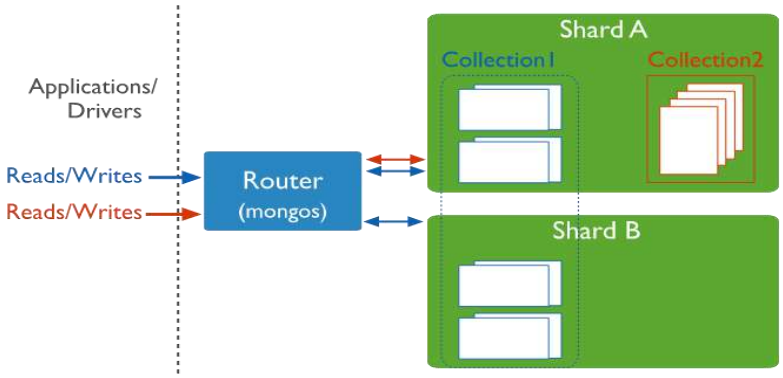


Figure 4.7 Connecting to a Sharded Cluster

Sharding Strategy

MongoDB offers two sharding strategies to distribute data across sharded clusters:

Hashed Sharding: This strategy involves computing a hash of the shard key field's value. Each chunk receives an assigned range based on these hashed shard key values. When resolving queries using hashed indexes, MongoDB automatically computes these hashes, eliminating the need for applications to do so, as depicted in Figure 4.8.

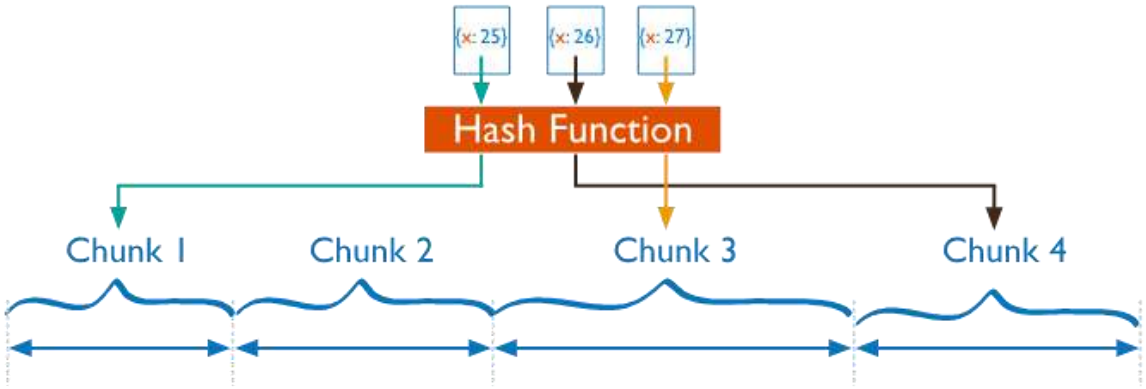


Figure 4.8 Hashed Sharding

Although a range of shard keys might be "close" to one another, their hashed values probably won't be in the same chunk. Distributing data based on hashed values promotes a more even data distribution, particularly in datasets where the shard key changes monotonically. However, with hashed distribution, range-based queries on the shard key are less apt to target a single shard. This can lead to increased cluster-wide broadcast operations.

Ranged Sharding : Ranged sharding divides data into ranges determined by the shard key values. Each chunk receives an assigned range based on these values, as shown in Figure 4.9.

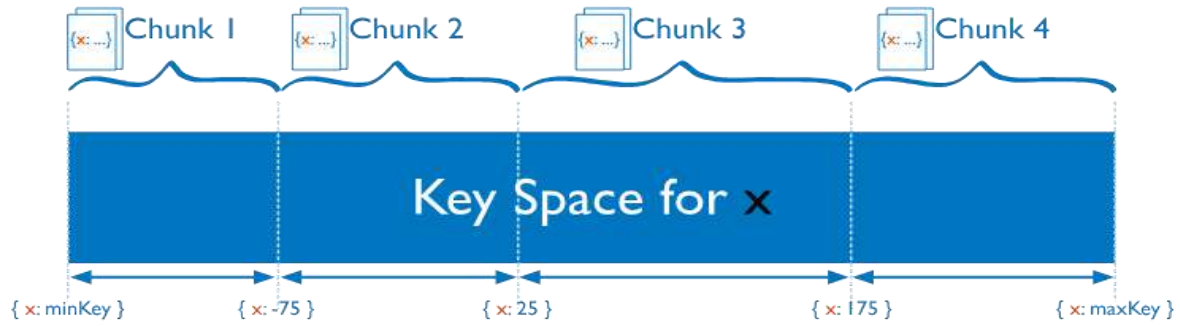


Figure 4.9 Ranged Sharding

Shard keys encompassing a range of values considered 'close' are likely to be situated within the same chunk. This feature streamlines targeted operations, allowing a mongos to direct operations exclusively to the shards containing the required data. However, the effectiveness of ranged sharding depends significantly on the selected shard key. Inappropriate choices of shard keys may result in imbalanced data distribution, undermining the benefits of sharding and potentially causing performance bottlenecks. For further insights into range-based sharding, please refer to the section on shard key selection.

Zones in Sharded Clusters

Zones can enhance data locality for sharded clusters spanning multiple data centers. In sharded clusters, zones of sharded data can be established based on the shard key. Each zone can be linked to one or more shards within the cluster. Moreover, a single shard can be associated with multiple zones. In a balanced cluster, MongoDB ensures that chunks within a specific zone migrate only to the shards associated with that zone.

Each zone encompasses one or more ranges of shard key values. The range within a zone is always inclusive of its lower boundary and exclusive of its upper **boundary** and a sample zones in sharded clusters is depicted in Figure 4.10.

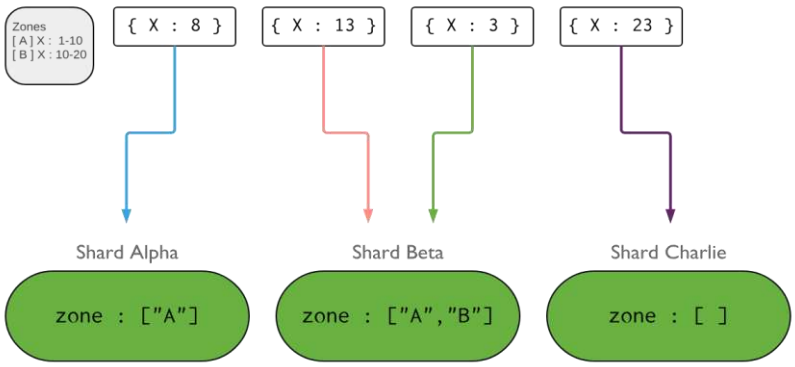


Figure 4.10 Zones in Sharded Clusters

When defining a new range for a zone's coverage, it's essential to use fields from the shard key. If a compound shard key is in use, the range should include its prefix. Consideration of potential future zone usage is crucial when choosing a shard key. Setting up zones and zone ranges prior to sharding an empty or non-existent collection allows for a more efficient initialization of zoned sharding.

Collations in Sharding

To shard a collection with a default collation, use the shardCollection command combined with the collation: { locale: "simple" } option. For successful sharding, the following conditions must be met:

1. The collection should have an index with the shard key as its prefix.
2. This index must possess the collation { locale: "simple" }.

When establishing new collections with a collation, it's essential to meet these conditions before sharding the collection.

Change Streams

Change streams are available for both replica sets and sharded clusters. They provide applications with the capability to capture real-time data changes, sidestepping the complexities and potential risks of tailing the oplog. Applications can utilize change streams to subscribe to data modifications in a single collection or across multiple collections.

Transactions

With the advent of distributed transactions, multi-document transactions became possible on sharded clusters. Changes within a transaction are not visible to outside operations until that

transaction is finalized. However, when multiple shards are impacted by a transaction, not every external read operation waits for the transaction's outcome to be observable on all shards. For example, if a completed transaction has Write 1 showing on Shard A, but Write 2 is still not evident on Shard B, an outside read operation using a "local" read concern might see the outcome of Write 1 but not Write 2.

4.8 MongoDB Replication

MongoDB's architecture and features that relate to maintaining read availability during replica set changes:

1.Read Preference: MongoDB allows you to specify read preferences for your queries. You can specify whether a query should target the primary, secondary, or a combination of both. By using secondary read preferences, you can distribute read traffic to replica set members, even during primary failovers. This can help maintain read availability.

2.Oplog: MongoDB uses an oplog (short for "operations log") to record all write operations. This oplog is continuously replicated to secondary members. During failovers, the new primary uses the oplog to catch up with the changes made to the data. This helps ensure that reads on the new primary are consistent with the last write operation.

3.Automatic Failover: When a primary node fails in a MongoDB replica set, the remaining healthy nodes automatically initiate an election to select a new primary. This process is designed to minimize downtime and ensure that one of the secondary nodes takes over the role of the primary. This enables uninterrupted read and write operations.

A replica set in MongoDB is a group of mongod processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the foundation for all production deployments. Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication offers a level of fault tolerance against the loss of a single database server.

In certain scenarios, replication can enhance read capacity, allowing clients to distribute read operations across different servers. Maintaining copies of data in various data centers can improve data locality and availability for distributed applications. Additionally, one can establish extra copies for specific purposes, such as disaster recovery, reporting, or backup.

A replica set comprises a cluster of mongod instances that collectively maintain an identical dataset. It consists of multiple data-bearing nodes and, optionally, an arbiter node. Within the data-bearing nodes, one member exclusively holds the primary role, while the remaining nodes serve as secondary nodes and the Figure 4.11 shows how the primary and secondary nodes interact. The primary node handles all incoming write operations. A replica set can only have a single primary node capable of confirming writes using the { w: "majority" } write concern. However, in certain circumstances, another mongod instance might transiently perceive itself as the primary node. The

primary node maintains a comprehensive record of all alterations made to its datasets in its operational log, known as the oplog.

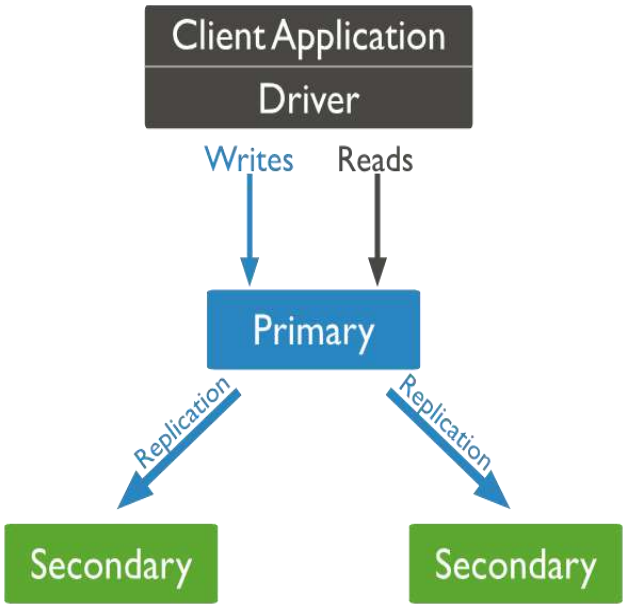


Figure 4.11 Replication in MongoDB

The secondaries replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set. If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary (Figure 4.12).

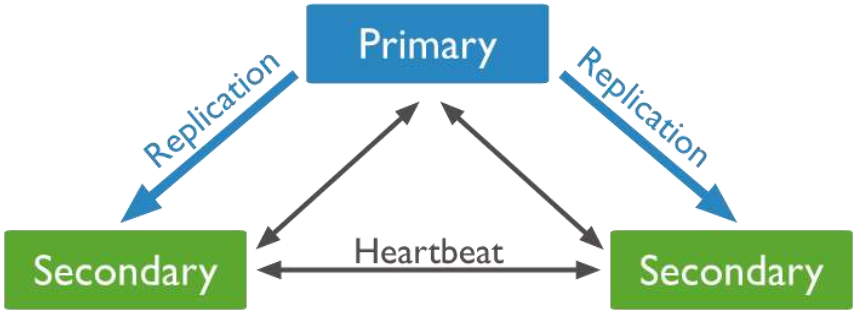


Figure 4.12 Secondary Holds Election for New Primary

In some circumstances (such as you have a primary and a secondary but cost constraints prohibit adding another secondary), you may choose to add a mongod instance to a replica set as an arbiter. An arbiter participates in elections but does not hold data (i.e. does not provide data redundancy) is shown in Figure 4.13.

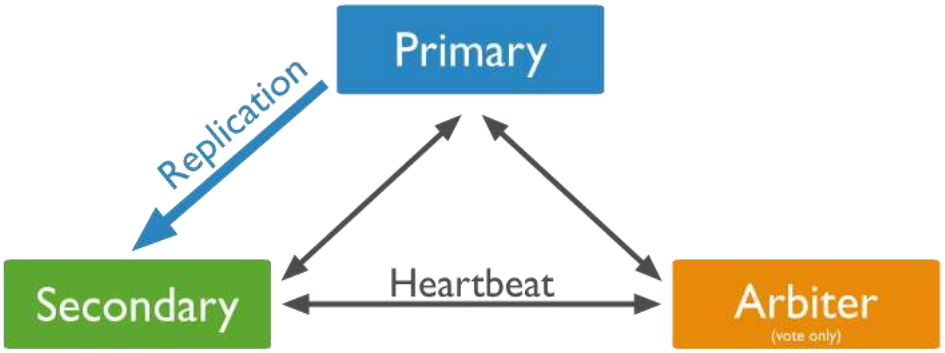


Figure 4.13 Arbiter Node

An arbiter will always be an arbiter whereas a primary may step down and become a secondary and a secondary may become the primary during an election.

Asynchronous Replication

The secondaries replicate the primary's oplog and asynchronously apply the operations to their datasets. By having the secondaries' data sets reflect the primary's data set, the replica set can continue to function despite the failure of one or more members.

Replication Lag and Flow Control

Replication lag indicates the duration required to replicate a write operation from the primary to a secondary. A minor delay may be tolerable, but escalating replication lag can lead to various issues, such as increased cache pressure on the primary node. In MongoDB 4.2, administrators have the ability to control the rate at which the primary applies its writes. This control aims to maintain the majority committed lag below a customizable maximum value, known as 'flowControlTargetLagSeconds.

Automatic Failover

If a primary node fails to communicate with other members of the set for a duration exceeding the configured electionTimeoutMillis period (defaulting to 10 seconds), an eligible secondary initiates an election to declare itself as the new primary. The cluster endeavors to complete the election process and reinstate normal operations by selecting a new primary (Figure 4.14).

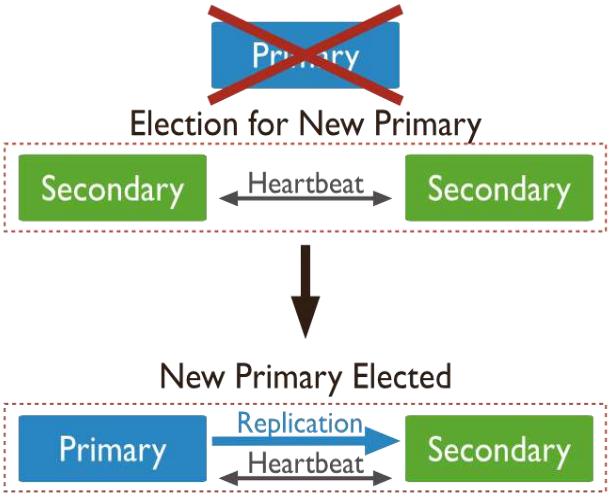


Figure 4.14 Election Process

The replica set cannot process write operations until the election completes successfully. However, the replica set can continue to serve read queries if such queries are configured to run on secondary's while the primary is offline. The median time before a cluster elects a new primary should not typically exceed 12 seconds, assuming default replica configuration settings.

This includes time required to mark the primary as unavailable and call and complete an election. Factors such as network latency may extend the time required for replica set elections to complete, subsequently impacting the duration during which a cluster can function without a primary. These factors are dependent on your particular cluster architecture.

4.9 Web Application Development using MongoDB with PHP and JAVA

Web application development using MongoDB with both PHP and Java is a powerful approach, enabling the harnessing of the strengths inherent in both languages and MongoDB's adaptable NoSQL database. To set up your development environment, start by installing MongoDB, PHP, and the Java development environment on your local machine or a server. Once your environment is prepared, outline the structure of your data in MongoDB, encompassing the design of the database schema, collections, and documents. In both PHP and Java, establishing a connection to your MongoDB database necessitates using the respective MongoDB drivers or libraries provided for each language. This connection is the foundation for interacting with your MongoDB data.

Within the chosen PHP environment, create backend logic, define routes, build controllers and models, and make use of the MongoDB PHP library to facilitate interactions with the database. This phase will also involve the implementation of essential components, including user authentication, business logic, and the creation of API endpoints. When working with Java, have the flexibility to select a Java framework that suits the project, such as Spring Boot or Java EE, or choose to build a Java-based backend from scratch. In the Java environment, develop RESTful

APIs that handle data interactions with MongoDB. To connect to the database, leverage MongoDB Java drivers. Additionally, implement various features, incorporate business logic, and establish security measures to safeguard the web application. This collaborative effort in both PHP and Java ensures that the backend operates efficiently and securely.

4.9.1 MongoDB PHP Driver

To work with MongoDB in PHP, one can integrate the MongoDB PHP Driver into their application. This driver consists of two essential components:

- The extension, which provides a low-level API and mainly serves to integrate libmongoc and libbson with PHP.
- The library, which provides a high-level API for working with MongoDB databases consistent with other MongoDB language drivers.

Driver Architecture

This section outlines the interaction between the components of the PHP driver. These components are categorized as follows:

- High-Level API, which encompasses the library and other integrations.
- Extension, involving the component that integrates with system libraries.
- System, which includes the C Driver, BSON library, and encryption library.

The following diagram (Figure 4,15) illustrates the architecture of the PHP driver components:

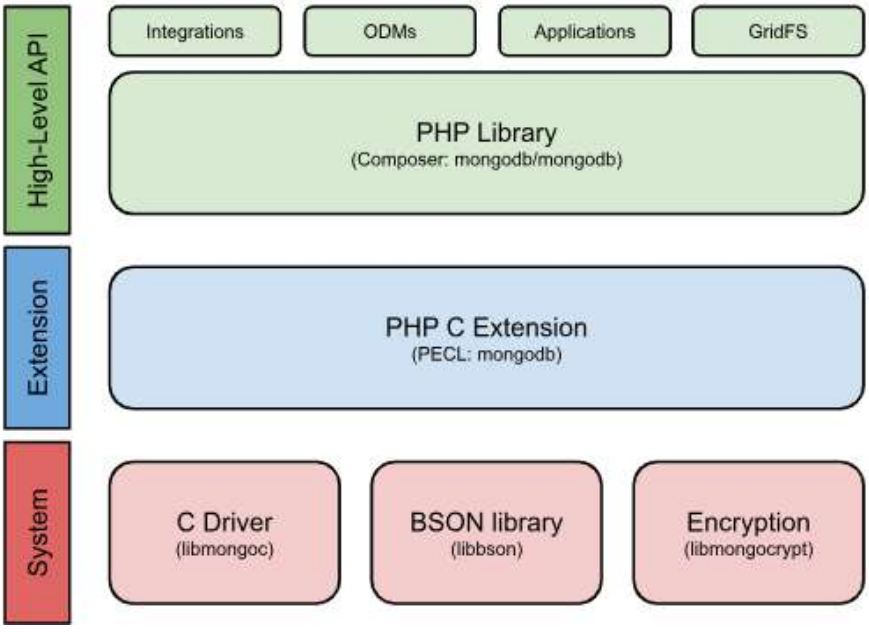


Figure 4.15 Architecture of the PHP driver components

The PHP library provides an API that is consistent with the other MongoDB drivers. The library is continually updated to meet cross-driver specifications.

The extension is distributed by using PECL and connects PHP to the system libraries. The extension's public API provides the following functionality:

- Connection management
- BSON encoding and decoding
- Object document serialization
- Command execution
- Cursor management

Steps for setting up the MongoDB PHP driver

- Download the MongoDB PHP Driver:
  - Visit the official MongoDB PHP driver repository to download the latest release.
- Install the MongoDB PHP Extension:
  - For Windows systems:
    - After downloading the appropriate file (mongodb.dll), place it in the PHP extensions directory.
    - Add the extension in your php.ini file:
      extension=php_mongodb.dll
  - Verify Installation:
    - Use php -m on the command line to check if the mongodb extension is loaded properly.
  - Connecting to MongoDB:

```
<?php
    $mongoClient = new MongoDB\Client("mongodb://localhost:27017");
```

After the extension is added, follow the PHP code to connect to MongoDB using the MongoDB\Client object.

Basic Operations

Once connected, various operations like insertion, retrieval, updating, and deletion can be performed:

- **Selecting a Database and Collection**

```
// Replace 'myDatabase' with your database name
$database = $mongoClient->myDatabase;
```

```
// Replace 'myCollection' with your collection name
   $collection = $database->myCollection;
```

- Inserting Data:
  ```
  $collection->insertOne(['name' => 'John', 'age' => 30]);
  ```

- Querying Data:
  ```
  $result = $collection->find(['name' => 'John']);
  foreach ($result as $entry) {
     var_dump($entry);
  }
  ```
- Updating Data
  ```
  $collection->updateOne(['name' => 'John'], ['$set' => ['age' => 31]]);
  ```

- Deleting Data:

  ```
  $collection->deleteOne(['name' => 'John']);
  ```

By following these steps and using the provided code snippets, one can easily set up the MongoDB PHP driver and perform basic operations in a PHP application connected to a MongoDB database.

Make a Connection and Select a Database

To establish a connection, specify the database name. If the specified database doesn't exist, MongoDB creates it automatically.

Below is the code snippet to connect to the database –

```
<?php
   // connect to mongodb
   $m = new MongoClient();


   echo "Connection to database successfully";
   // select a database
   $db = $m->mydb;


   echo "Database mydb selected";
?>
```

After the program executes, it will generate the following result:

```
Connection to database successfully
Database mydb selected
```

Create a Collection

Here is the code snippet to create a collection:

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->createCollection("mycol");
   echo "Collection created succsessfully";
?>
```

Upon execution, the program will yield the following result:

```
Connection to database successfully
Database mydb selected
Collection created succsessfully
```

Insert a Document

The insert() method is utilized to insert a document into MongoDB.

Here is the code snippet for inserting a document:

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   $document = array(
      "title" => "MongoDB",
      "description" => "database",
      "likes" => 100,
      "url" => "http://www.tutorialspoint.com/mongodb/",
      "by" => "tutorials point"
   );

   $collection->insert($document);
   echo "Document inserted successfully";
?>
```

Upon program execution, it will generate the following result:

```
Connection to the database successful.
Database 'mydb' selected.
Collection successfully chosen.
Document insertion successful.
```

Find All Documents

The find() method is employed to select all documents from the collection.

Here's the code snippet for selecting all documents:

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succsessfully";
  $cursor = $collection->find();
  // iterate cursor to display title of documents

  foreach ($cursor as $document) {
    echo $document["title"] . "\n";
  }
?>
```

Upon program execution, it will generate the following result:

```
Connection to the database successful.
Database 'mydb' selected.
Collection successfully chosen:
{
"title": "MongoDB"
}
```

Update a Document

To update a document, use the update() method.

In this example, the goal is to update the title of the inserted document to "Tutorial". Here's the code snippet to achieve this:

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";
   // now update the document
   $collection->update(array("title"=>"MongoDB"),
      array('$set'=>array("title"=>"Tutorial")));
   echo "Document updated successfully";

   // now display the updated document
   $cursor = $collection->find();

   // iterate cursor to display title of documents
   echo "Updated document";

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

Upon program execution, it will generate the following result:

```
Connection to the database successful.
Database 'mydb' selected.
Collection successfully chosen.
Document updated successfully.
Updated document:
{
"title": "Tutorial"
}
```

Delete a Document

To delete a document, use the remove () method.

In this example, the goal is to remove documents with the title "Tutorial". Here's the code snippet to achieve this:

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   // now remove the document
   $collection->remove(array("title"=>"Tutorial"),false);
   echo "Documents deleted successfully";

   // now display the available documents
   $cursor = $collection->find();

   // iterate cursor to display title of documents
   echo "Updated document";

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

Upon program execution, it will generate the following result:

```
Connection to the database successful.
Database 'mydb' selected.
Collection successfully chosen.
Documents deleted successfully.
```

Developing web applications with MongoDB using PHP requires integrating the MongoDB PHP Driver. The process begins by installing the driver, establishing a connection, and executing various operations like database and collection creation, document insertion, querying, updating, and deletion. The driver components, comprising the extension and library, assume vital roles in facilitating interactions with MongoDB. This approach offers a sturdy foundation for web development, harnessing the strength and adaptability of MongoDB's NoSQL database within PHP-based applications

4.9.2 MongoDB with JAVA

Once MongoDB is installed and running, you can interact with it using Java by using the MongoDB Java Driver. You'll need to add the MongoDB Java driver to your Java project to interact with the MongoDB database.

Java and MongoDB Connection:

Here's a basic example of how to connect to MongoDB from a Java program using the MongoDB Java Driver:

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class ConnectToDB {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser",
"myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      System.out.println("Credentials ::"+ credential);
   }
}
```

Ensure you have the necessary Java libraries and dependencies for the MongoDB Java driver configured in your Java project to interact with MongoDB from Java code.

On execution, the above program yields the following output:

```
Credentials: MongoCredential{
  mechanism = null,
  userName = 'sampleUser',
  source = 'myDb',
  password = <hidden>,
  mechanismProperties = {}
}
```

Connected to the database successfully.

This setup allows you to connect to MongoDB from your Java programs, enabling you to perform various database operations.

Create a Collection

To create a collection, the createCollection() method of the com.mongodb.client.MongoDatabase class is used.

Below is the code snippet to create a collection:

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class CreatingCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      //Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      //Creating a collection
      database.createCollection("sampleCollection");
      System.out.println("Collection created successfully");
   }
}
```

The output upon compiling the aforementioned program would be as follows:

```
Connected to the database successfully.
Collection created successfully.
```

Getting/Selecting a Collection

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class selectingCollection {
  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" , 27017 );

    // Creating Credentials
    MongoCredential credential;
    credential = MongoCredential.createCredential("sampleUser", "myDb",
      "password".toCharArray());
    System.out.println("Connected to the database successfully");
    // Accessing the database
    MongoDatabase database = mongo.getDatabase("myDb");

    // Creating a collection
    System.out.println("Collection created successfully");
    // Retrieving a collection
    MongoCollection<Document> collection = database.getCollection("myCollection");
}

    System.out.println("Collection myCollection selected successfully");
  }
```

To retrieve or select a collection from the database, the getCollection() method of the com.mongodb.client.MongoDatabase class is used. Below is the program to access a collection:

Upon compiling the given program, the expected result would be:

```
Connected to the database successfully.
Collection created successfully.
Collection 'myCollection' selected successfully.
```

Insert a Document

In MongoDB, you can utilize the insert() method of the com.mongodb.client.MongoCollection class to insert a document.

The code snippet below demonstrates document insertion:

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
public class InsertingDocument {
        public static void main( String args[] ) {

        // Creating a Mongo client
        MongoClient mongo = new MongoClient( "localhost" , 27017 );

        // Accessing the database
        MongoDatabase database = mongo.getDatabase("myDb");

        // Creating a collection
        database.createCollection("sampleCollection");
        System.out.println("Collection created successfully");

        // Retrieving a collection
        MongoCollection<Document> collection =
database.getCollection("sampleCollection");
        System.out.println("Collection sampleCollection selected
successfully");
        Document document = new Document("title", "MongoDB")
        .append("description", "database")
        .append("likes", 100)
        .append("url", "http://www.tutorialspoint.com/mongodb/")
        .append("by", "tutorials point");

        //Inserting document into the collection
```

```
            collection.insertOne(document);
            System.out.println("Document inserted successfully");
}
```

Upon compiling the provided program, the anticipated output would be:

```
Connected to the database successfully.
Collection 'sampleCollection' selected successfully.
Document inserted successfully.
```

Retrieve All Documents

To retrieve all documents from a collection, the find() method of the com.mongodb.client.MongoCollection class is used. This method returns a cursor, which necessitates iteration to access the documents. The code snippet below illustrates selecting all documents:

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class RetrievingAllDocuments {
        public static void main( String args[] ) {

        // Creating a Mongo client
        MongoClient mongo = new MongoClient( "localhost" , 27017 );
// Creating Credentials
```

```
        MongoCredential credential;
        credential = MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
        System.out.println("Connected to the database successfully");

        // Accessing the database
        MongoDatabase database = mongo.getDatabase("myDb");

        // Retrieving a collection
MongoCollection<Document> collection =
database.getCollection("sampleCollection");
System.out.println("Collection sampleCollection selected successfully");
Document document1 = new Document("title", "MongoDB")
.append("description", "database")
.append("likes", 100)
.append("url", "http://www.tutorialspoint.com/mongodb/")
.append("by", "tutorials point");
```

```
        Document document2 = new Document("title", "RethinkDB")
        .append("description", "database")
        .append("likes", 200)
        .append("url", "http://www.tutorialspoint.com/rethinkdb/")
                .append("by", "tutorials point");
        List<Document> list = new ArrayList<Document>();
        list.add(document1);
        list.add(document2);
        collection.insertMany(list);
        // Getting the iterable object
        FindIterable<Document> iterDoc = collection.find();
        int i = 1;
        // Getting the iterator
        Iterator it = iterDoc.iterator();
        while (it.hasNext()) {
                System.out.println(it.next());
                i++;
                }
        }
}
```

The output upon compiling the provided program would likely be as follows:

```
Connected to the database successfully.

Collection 'sampleCollection' selected successfully.

Document{_id=5dce4e9ff68a9c2449e197b2, title=MongoDB,
description=database, likes=100, url=http://www.tutorialspoint.com/mongodb/,
by=tutorials point}

Document{_id=5dce4e9ff68a9c2449e197b3, title=RethinkDB,
description=database, likes=20}
```

Update Document

To update a document in the collection, the updateOne() method of the com.mongodb.client.MongoCollection class is utilized. The code snippet provided aims to update the first document.

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class UpdatingDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser",
"myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      // Retrieving a collection
      MongoCollection<Document> collection =
database.getCollection("sampleCollection");
      System.out.println("Collection myCollection selected successfully");
```

```
            collection.updateOne(Filters.eq("title", 1), Updates.set("likes",
        150));
            System.out.println("Document update successfully...");


            // Retrieving the documents after updation
            // Getting the iterable object
            FindIterable<Document> iterDoc = collection.find();
            int i = 1;
            // Getting the iterator
            Iterator it = iterDoc.iterator();
            while (it.hasNext()) {
              System.out.println(it.next());
              i++;
            }
          }
}
```

The provided program, upon compilation, should output:

```
Connected to the database successfully.
Collection 'myCollection' selected successfully.
Document updated successfully.
Document{_id=5dce4e9ff68a9c2449e197b2, title=MongoDB,
description=database, likes=100, url=http://www.tutorialspoint.com/mongodb/,
by=tutorials point}
Document{_id=5dce4e9ff68a9c2449e197b3, title=RethinkDB,
description=database, likes=20}
```

It appears to show successful connection and collection selection, along with the updated document outputs.

Delete a Document

To remove a document from a collection, you'll utilize the deleteOne() method within the com.mongodb.client.MongoCollection class. The following code demonstrates the deletion of a document:

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DeletingDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser",
"myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
```

```
            // Retrieving a collection
            MongoCollection<Document> collection =
        database.getCollection("sampleCollection");
            System.out.println("Collection sampleCollection selected
        successfully");
            // Deleting the documents
            collection.deleteOne(Filters.eq("title", "MongoDB"));
            System.out.println("Document deleted successfully...");

            // Retrieving the documents after updation
            // Getting the iterable object
            FindIterable<Document> iterDoc = collection.find();
            int i = 1;
            // Getting the iterator
            Iterator it = iterDoc.iterator();
            while (it.hasNext()) {
              System.out.println(it.next());
              i++;
            }
          }
}
```

On compiling, the above program gives you the following result –

```
Connected to the database successfully.
Collection 'sampleCollection' selected successfully.
Document deleted successfully...
Document{_id=5dce4e9ff68a9c2449e197b3, title=RethinkDB,
description=database, likes=20}
```

Dropping a Collection

To drop a collection, you'd actually use the drop() method from the com.mongodb.client.MongoCollection class. Here's an example code to delete a collection:

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DropingCollection {

  public static void main( String args[] ) {
    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" , 27017 );
    // Creating Credentials
    MongoCredential credential;
    credential = MongoCredential.createCredential("sampleUser",
"myDb",
      "password".toCharArray());
    System.out.println("Connected to the database successfully");

    // Accessing the database
    MongoDatabase database = mongo.getDatabase("myDb");

    // Creating a collection
    System.out.println("Collections created successfully");
    // Retrieving a collection
    MongoCollection<Document> collection =
database.getCollection("sampleCollection");
    // Dropping a Collection
    collection.drop();
    System.out.println("Collection dropped successfully");
  }
```

```
}
```

Upon compiling the given program, the expected output would be:

```
Connected to the database successfully.
Collection 'sampleCollection' selected successfully.
Collection dropped successfully
```

Listing All the Collections

To list all collections within a database, the listCollectionNames() method of the com.mongodb.client.MongoDatabase class is used. The following code demonstrates listing all collections within a database:

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class ListOfCollection {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" , 27017 );
    // Creating Credentials
    MongoCredential credential;
    credential = MongoCredential.createCredential("sampleUser",
"myDb",
      "password".toCharArray());
    System.out.println("Connected to the database successfully");

    // Accessing the database
    MongoDatabase database = mongo.getDatabase("myDb");
    System.out.println("Collection created successfully");
    for (String name : database.listCollectionNames()) {
      System.out.println(name);
    }
  }
}
```

Upon compiling the provided program, the expected output would likely be:

```
Connected to the database successfully.
Collection created successfully.
myCollection
myCollection1
myCollection5
```

4.10 Self-Assessment questions and Exercises

1. Define and differentiate between NoSQL and SQL databases.
2. List the key characteristics of NoSQL databases.
3. Explain scenarios where NoSQL databases outperform SQL databases.
4. Define the CAP theorem and its implications for database systems.
5. Explain the trade-offs among Consistency, Availability, and Partition Tolerance in database systems.
6. Discuss the considerations and challenges involved in migrating from a relational database to a NoSQL database.
7. Highlight the differences in data modeling and querying approaches between RDBMS and NoSQL.
8. Describe CRUD operations (Create, Read, Update, Delete) in MongoDB.
9. Provide examples of each CRUD operation using MongoDB.
10. Explain the concept of sharding in MongoDB.
11. Discuss the benefits and challenges of sharding in a distributed database environment.
12. Define replication in the context of MongoDB.
13. Explain the purpose and advantages of replication in a database system.
14. Define replication in the context of MongoDB.
15. Explain the purpose and advantages of replication in a database system.

4.11 Summary

The shift from SQL to NoSQL databases embodies significant differences, notably illuminated by the CAP Theorem, driving migration considerations from RDBMS to NoSQL solutions. MongoDB, a popular NoSQL option, stands out, offering flexibility and scalability. CRUD operations, sharding, and replication mechanisms within MongoDB ensure efficient data handling, providing reliability and performance at scale. Its integration with PHP and Java in web application development provides a powerful, flexible, and scalable environment for modern applications, highlighting the adaptability and utility of MongoDB in diverse development landscapes.

Keywords

CAP Theorem, ACID, Base, Consistency, Availability, Partition tolerance, Migrating, RDBMS, NOSQL, MongoDB, CRUD Operations, Sharding, Replication, Web Application Development, PHP, Java

4.12 Further readings

Books:

1. MongoDB: The Definitive Guide" by Kristina Chodorow.
2. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" by Pramod J. Sadalage and Martin Fowler.
3. https://www.infoq.com/articles/mongodb-java-php-python/
4. https://www.mongodb.com/docs/manual

UNIT 5
DATA WAREHOUSING AND DATA MINING

CONTENT

5.1 Introduction

In today's data-driven world, organizing and interpreting vast amounts of information are crucial for businesses and researchers. Enter the realm of data warehousing and data mining. A **data warehouse** is a central repository that consolidates, transforms, and provides data from various sources for analysis. It features a three-tier architecture: data source layer, the data warehouse layer, and the presentation layer. A key component in this system is the **data cube**, enabling a multi-dimensional representation of data that allows for intricate querying.

Diving deeper into the analytical side, there is a distinction between **Online Analytical Processing (OLAP)** and **Online Transaction Processing (OLTP)**. While OLTP handles everyday transactional activities, OLAP is designed for complex querying and reporting. It often leverages specific schemas such as the **Star Schema**, **Snow Flake Schema**, and **Fact Constellation Schema**. Furthermore, in scenarios where a comprehensive data warehouse might be overwhelming or unnecessary, a **Data Mart** offers a more focused and streamlined subset of the warehouse, catering to specific business functions.

Moving beyond the warehousing, D**ata Mining** comes into play as the process of discovering patterns, correlations, and insights from large datasets. Techniques such as the **Apriori Algorithm** are instrumental in association rule mining while **Decision Trees** are used for data classification, specifically those that capitalize on information gain. Lastly, for dividing data into specific clusters based on similarity, the **k-means clustering** method is a popular choice. Together, data warehousing and data mining provide a robust toolkit for organizations to harness the power of their data, driving insights and informed decision-making.

5.2 Learning Objectives

1. Understanding the fundamental of Data Warehousing
2. Distinguishing OLAP and OLTP
3. Mastering Data Modeling Techniques in OLAP
4. Introduction to Data Marts
5. Basics of Data Mining
6. Learning Classification Techniques
7. Understanding Clustering Algorithms

5.3 Overview Data Warehousing and Data Mining

Data Warehousing and Data Mining are foundational in data management and analysis. Data Warehousing with its three-tier architecture, centralizes data, consolidates, transforms, and stores it, utilizing constructs like the Data Cube. This storage system draws a line between Online Analytical Processing (OLAP) - for intricate queries, and Online Transaction Processing (OLTP) - for routine transactions. In OLAP, organizational structures such as the Star Schema, Snow Flake

Schema, and Fact Constellation Schema are employed. Complementing this is the Data Mart, a focused subset of the data warehouse tailored to specific business needs.

Concurrently, Data Mining explores vast data sets to uncover hidden patterns and relationships. The Apriori Algorithm is pivotal for association rule mining, revealing interconnected data elements. Decision Tree Induction, especially when bolstered with information gain, is the preferred method for classification. The k-means clustering is a popular choice for dataset segmentation. Together, Data Warehousing and Data Mining ensure efficient storage and insightful analysis of data, serving as twin pillars in modern data management.

5.4 Data Warehouse

A data warehouse is a centralized repository designed to store, consolidate, and retrieve data from various sources to support business intelligence (BI) activities. Its primary purpose is to transform raw data into meaningful and useful information for analytical processing, decision-making, and reporting. Unlike typical databases are optimized for transactional processes, data warehouses are optimized for querying and reporting. They are structured to aggregate data from different operational systems such as CRM, ERP, or financial systems, thus offering a unified view of the organization's data.

One prominent feature of data warehouses is their use of dimensional modeling, which includes facts and dimensions, to ensure data is organized efficiently. The ETL (Extract, Transform, Load) process plays a vital role, where data is extracted from source systems, transformed into a consistent format, and then loaded into the data warehouse. Over time, data warehousing has evolved, leading to the advent of Data Lakes and more real-time processing solutions. Nonetheless, data warehouses remain crucial for businesses that need historical data analysis, trend spotting, and strategic decision-making.

Characteristics

The key characteristics of a data warehouse:

1. Subject-Oriented: Data warehouses are organized around major subjects of an organization, such as customers, products, sales, etc., rather than the major application areas.
2. Integrated: Data in a data warehouse is sourced from various operational systems and processed to ensure consistency in naming conventions, measurement, encoding, etc.
3. Time-Variant: Data is stored in a way to allow comparisons over time. This historical perspective can span from a few months to several years, unlike operational systems that typically store only recent data.
4. Non-Volatile: Data in the data warehouse doesn't change once it's stored, ensuring stability in analytical reporting and providing a consistent, time-variant snapshot of data, regardless of changes in source systems.
5. Large Volume: Data warehouses typically store extensive amounts of data, accumulating it from diverse sources over extended periods.
6. Supports Analytical Processing: Unlike operational databases designed for CRUD (Create, Read, Update, Delete) operations, data warehouses are structured to support complex queries and analytical processing, facilitating trend analysis and decision-making.

7. Separate from Operational Systems: Data warehouses are kept separate from operational databases for performance and safety reasons, preventing complex queries from slowing down operational systems.
8. Schema Design: Data warehouses commonly use dimensional modeling with star or snowflake schemas, which streamline query processing.
9. Data Quality and Consistency: Through the ETL (Extract, Transform, Load) process, data within a data warehouse is cleansed, transformed, and loaded, ensuring high quality and consistency.
10. Granularity: Refers to the level of detail or summarization of the data. A data warehouse will often store both detailed (granular) and summarized data.
11. Scalability: Data warehouses need to be scalable to accommodate the increasing volume of data.

The exact characteristics and functionalities of a data warehouse can vary depending on its design, the tools in use, and specific business requirements.

Data warehouse architecture refers to the design and layout of the structural framework used to store, manage, and retrieve data. While the specific architecture can vary based on organizational needs, there is a general framework that is commonly adopted. Here's an overview of a typical data warehouse architecture:

1. Data Source:
    - Operational Databases: These are transactional databases like CRM, ERP, and other OLTP systems
    - Flat Files: These could be logs, CSVs, or other unstructured/semi-structured data.
    - External Data Sources: Data from third-party vendors, cloud sources, or web services.
2. Staging Area:
    - Before loading into the data warehouse, data is temporarily held in a staging area.
    - This is where initial cleansing and transformation occurs to convert raw data into a format suitable for the data warehouse.
3. ETL Process (Extract, Transform, Load):
    - Extract: Data is extracted from the source systems.
    - Transform: Data is transformed into the required format, and inconsistencies are resolved. This can include operations like cleansing, aggregations, enrichments, etc.
    - Load: Data is loaded into the data warehouse. Depending on requirements, this can be a bulk load, incremental load, or real-time load.
4. Data Warehouse Storage:
    - Fact Tables: Contain metrics or measurements of the business process. It includes foreign keys to dimension tables.
    - Dimension Tables: Store attributes or dimensions by which the facts can be viewed or sliced-and-diced.
    - These tables typically follow a schema like star schema or snowflake schema.

5. Data Marts:
    - These are subsets of data warehouses and focus on specific business areas, such as sales, finance, or marketing.
    - They make data access faster and more specialized for end-users.
6. OLAP Servers (Online Analytical Processing):
    - They facilitate multi-dimensional analysis of data.
    - There are different OLAP architectures like ROLAP (Relational OLAP), MOLAP (Multidimensional OLAP), and HOLAP (Hybrid OLAP).
7. Presentation Layer:
    - BI Tools: Tools that enable users to query and visualize data. Examples include Tableau, Power BI, and QlikView.
    - Dashboards and Reports: Displays summarized data, KPIs, and other metrics.
8. Metadata, Governance, and Management:
    - Metadata Repository: Contains data about the data, such as source, transformation applied, usage, and more.
    - Data Governance and Quality: Ensures data's integrity, quality, and security in the warehouse.
    - Management Tools: Help in monitoring, managing, and optimizing the data warehouse.
9. Data Lake (Modern Architectures):
    - Some modern architectures incorporate a data lake, which stores raw, granular data in its native format.
    - Data can be structured, semi-structured, or unstructured.

Remember, the exact structure and components can vary based on factors like the size of the organization, the volume of data, technology in use, and specific business needs. The broad perspective of standard components in a data warehouse architecture is shown in Figure 5.1

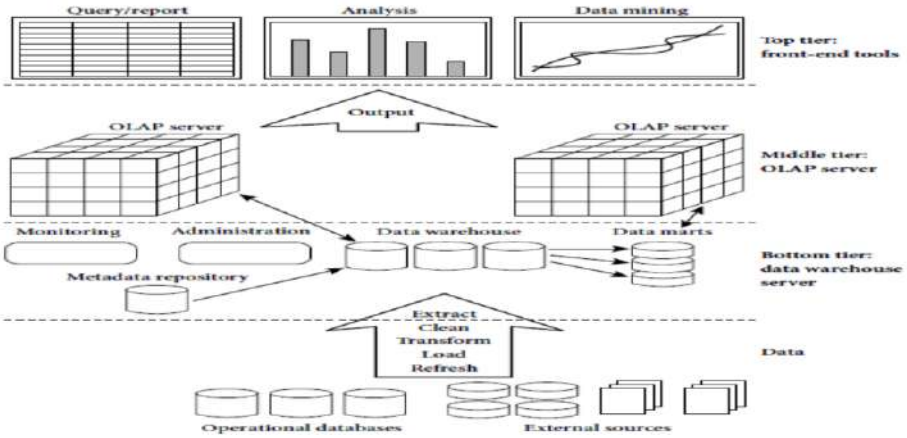


Figure 5.1 Three Tier Date Warehouse Architecture

5.4.1 Online Transaction Processing

OLTP is an operational system that supports transaction-oriented applications in a 3-tier architecture. It handles an organization's day-to-day transactions and is primarily focused on query processing, maintaining data integrity in multi-access environments. Its efficiency is typically measured by the total number of transactions processed per second. The acronym OLTP stands for Online Transaction Processing.

Characteristics of OLTP

The important characteristics of OLTP:

- OLTP uses transactions that include small amounts of data.
- Indexed data in the database can be accessed easily.
- OLTP has a large number of users.
- It has fast response times
- Databases are directly accessible to end-users
- OLTP uses a fully normalized schema for database consistency.
- The response time of OLTP system is short.
- It strictly performs only the predefined operations on a small number of records.
- OLTP stores the records of the last few days or a week.
- It supports complex data models and tables.

Type of queries that an OLTP system can Process:

OLTP system is an online database changing system. Therefore, it supports database query such as insert, update, and delete information from the database.

Consider a point of sale system of a supermarket, following are the sample queries that this system can process:

- Retrieving the description of a particular product.
- Filtering all products related to the supplier.
- Searching the record of the customer.
- Listing products having a price less than the expected amount.

Architecture of OLTP

1. Business / Enterprise Strategy: Enterprise strategy deals with the issues that affect the organization as a whole. In OLTP, it is typically developed at a high level within the firm, by the board of directors or the top management
2. Business Process: OLTP business process is a set of activities and tasks that, once completed, will accomplish an organizational goal.

3. Customers, Orders, and Products: OLTP database store information about products, orders (transactions), customers (buyers), suppliers (sellers), and employees.
4. ETL (Extract, Transform and Load) Processes: It separates the data from various RDBMS source systems, then transforms the data (like applying concatenations, calculations, etc.) and loads the processed data into the Data Warehouse system.
5. Data Mart and Data warehouse: A data mart is a structure/access pattern specific to data warehouse environments. It is used by OLAP to store processed data.
6. Data Mining, Analytics, and Decision Making: Data stored in the data mart and data warehouse can be used for data mining, analytics, and decision making.

This data helps us discover data patterns, analyze raw data, and make analytical decisions for an organization's growth.

Example of OLTP Transaction

Consider an example of the OLTP system, such as an ATM center. Assume a couple with a joint bank account who both arrive at different ATM centers simultaneously, intending to withdraw the total amount from their account. However, only the person who completes the authentication process first will be able to withdraw the money. In this case, the OLTP system makes sure that the withdrawn amount will be never more than the amount present in the bank. The key to note here is that OLTP systems are optimized for transactional superiority instead of data analysis.

Other examples of OLTP system are:

- Online banking
- Online airline ticket booking
- Sending a text message
- Order entry
- Add a book to shopping cart

Advantages of OLTP:

Following are the pros/benefits of OLTP system:

- OLTP offers accurate forecast for revenue and expense.
- It provides a solid foundation for a stable business /organization due to timely modification of all transactions.
- OLTP makes transactions much easier on behalf of the customers.
- It broadens the client base for an organization by speeding up and simplifying individual processes.
- OLTP provides support for bigger databases.
- Partition of data for data manipulation is easy.
- We need OLTP to use the tasks which are frequently performed by the system.

- When we need only a small number of records.
- The tasks that include insertion, updation, or deletion of data.
- It is used when you need consistency and concurrency in order to perform tasks that ensure its greater availability.

Disadvantages of OLTP:

Here are cons/drawbacks of OLTP system:

- If the OLTP system faces hardware failures, then online transactions get severely affected.
- OLTP systems allow multiple users to access and change the same data at the same time, which many times created an unprecedented situation.
- If the server hangs for seconds, it can affect to a large number of transactions.
- OLTP required a lot of staff working in groups in order to maintain inventory.
- Online Transaction Processing Systems do not have proper methods of transferring products to buyers by themselves.
- OLTP makes the database much more susceptible to hackers and intruders.
- In B2B transactions, there are chances that both buyers and suppliers miss out efficiency advantages that the system offers.
- Server failure may lead to wiping out large amounts of data from the database.
- You can perform a limited number of queries and updates.

Challenges of an OLTP System

It allows more than one user to access and change the same data simultaneously. Therefore, it requires concurrency control and recovery technique in order to avoid any unprecedented situations

OLTP system data are not suitable for decision making. You have to use data of OLAP systems for "what if" analysis or the decision making.

5.4.2 Online Analytical Processing

Online Analytical Processing (OLAP) is a category of software that allows users to analyze information from multiple database systems at the same time. It is a technology that enables analysts to extract and view business data from different points of view. Analysts frequently need to group, aggregate and join data. These operations in relational databases are resource intensive. With OLAP data can be pre-calculated and pre-aggregated, making analysis faster. OLAP databases are divided into one or more cubes. The cubes are designed in such a way that creating and viewing reports become easy. OLAP stands for Online Analytical Processing.

A data warehouse extracts information from various data sources and formats, including text files, Excel sheets, and multimedia files. This extracted data is cleaned, transformed, and then loaded into an OLAP server or OLAP cube where information is pre-calculated for further analysis. A data cube is derived from a subset of attributes in the database, with specific attributes chosen as measure attributes , representing the values of interest. Other attributes are selected as dimensions or functional attributes. These measure attributes are then aggregated according to their respective dimensions, resulting in multidimensional matrices known as data cubes. These cubes consist of numeric facts, called measures, categorized by dimensions and are often referred to as hypercube. The overall process of the OLAP is depicted in Figure 5.2.

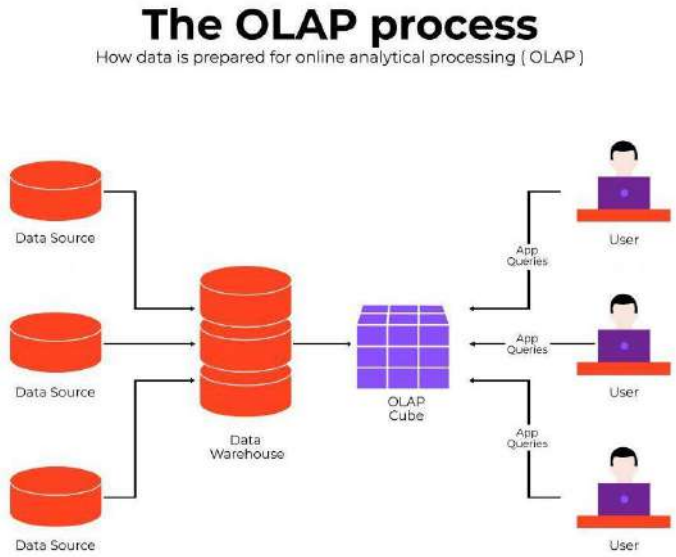


Figure 5.2 OLAP Process

In the multidimensional data model, the emphasis is on collecting numeric measures. Each measure relies on a set of dimensions. For instance, consider a running example based on sales data. The measure attribute in our scenario is "sales," and the dimensions are "Product," "Location," and "Time." For a given product, location, and time, there's a corresponding sales value. If we use unique identifiers like pid for a product, locid for location, and timeid for time, the sales information appears as a three-dimensional array labeled "Sales." This array can be visually represented, as shown in Figure 5.3. For clarity, we might display only the values for a specific locid value (e.g., locid=1), representing a slice orthogonal to the locid axis.

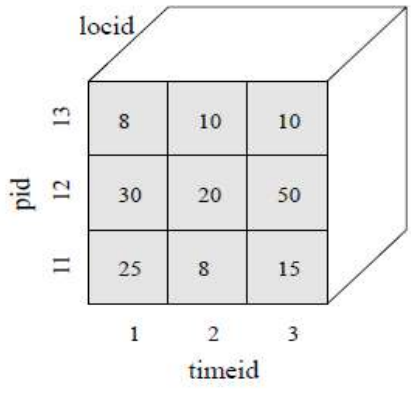


Figure 5.3 Sales: A Multidimensional Dataset

The data in a multidimensional array can also be represented as a relation, as illustrated in Figure 5.4, which shows the same data as in Figure 5.3, additional rows corresponding to the `slice' locid=2 are shown in addition to the data visible in Figure 5.4. The relation relates the dimensions to the measure of interest and is commonly referred to as the fact table.

Now, let's turn the attention to dimensions. Each dimension can have a set of associated attributes. For example, the Location dimension is identied by the locid attribute, which is used to identify a location in the Sales table. We will assume that it also has attributes country, state, and city. Similarly, the Product dimension is assumed to have attributes pname, category, and price, in addition to the identier pid. The category of a product indicates its general nature; for example, a product 'pants' could have category value apparel. We will assume that the Time dimension has attributes date, week, month, quarter, year, and holiday flag, in addition to the identifier timeid.

locid	*city*	*state*	*country*
1	Ames	Iowa	USA
2	Chennai	TN	India
5	Tempe	Arizona	USA

Locations

pid	*pname*	*category*	*price*
11	Lee Jeans	Apparel	25
12	Zord	Toys	18
13	Biro Pen	Stationery	2

Products

pid	*timeid*	*locid*	*sales*
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26
12	2	2	45
12	3	2	20
13	1	2	20
13	2	2	40
13	3	2	5

Sales

Figure 5.4 Locations, Products, and Sales Represented as Relations

For every dimension, the related values can be organized into a hierarchical structure. For instance, cities are part of states, while states fall under countries. Similarly, dates are categorized into weeks and months. Both weeks and months fall under quarters, and quarters are grouped into years. (It's important to note that a week might overlap into two months, so it's not strictly contained within a month.) Certain attributes within a dimension define its position concerning its inherent hierarchy. The hierarchies for the Product, Location, and Time are depicted in Figure 5.5 at the attribute level.

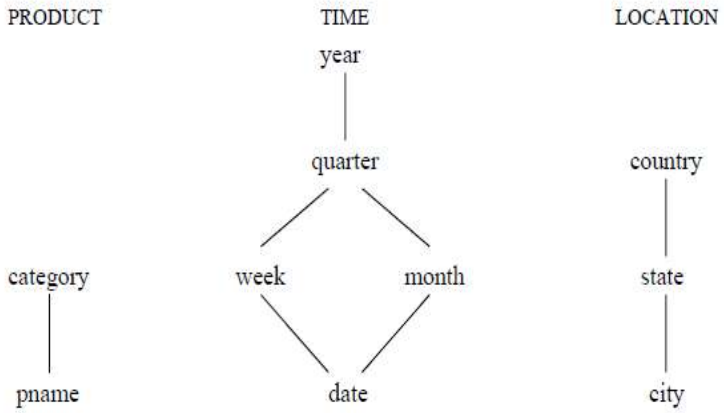


Figure 5.5 Dimension Hierarchies

Information about dimension also be represented as a collection of relations:

Locations(locid: integer, city: string, state: string, country: string)

Products(pid: integer, pname: string, category: string, price: real)

Times(timeid: integer, date: string, week: integer, month: integer, quarter: integer, year: integer, holiday flag: boolean)

For every dimension, the related values can be organized into a hierarchical structure. For instance, cities are part of states, while states fall under countries. Similarly, dates are categorized into weeks and months. Both weeks and months fall under quarters, and quarters are grouped into years. (It's important to note that a week might overlap into two months, so it's not strictly contained within a month.) Certain attributes of a dimension define its position concerning its inherent hierarchy. The hierarchies for the Product, Location, and Time in our example are depicted at the attribute level in the provided figure.

5.4.3 OLTP vs OLAP in Database Design

OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) are two fundamental classes of database systems, each designed to serve specific types of workloads. Here's a comparison of their key characteristics:

1. **Purpose:**
   - OLTP: Designed to manage day-to-day transactional data. Examples include sales transactions, customer details, order processing, and so on.
   - OLAP: Designed for complex data analysis and reporting. These databases are structured to answer business intelligence questions like trends, forecasting, and statistics.
2. **Database Design:**
   - OLTP: Typically uses a relational database design with entities and relationships, optimized for insert/update operations. It often involves normalization to avoid data redundancy.

- OLAP: Uses a star or snowflake schema in its database design, which is optimized for querying and report generation. It often involves denormalization and uses things like data cubes to pre-aggregate data.

3. **Data Volume:**
   - OLTP: Manages a large number of small transactions.
   - OLAP: Manages a lower volume of transactions, but queries can be very complex and might scan millions of rows.
4. **Query Complexity:**
   - OLTP: Short, atomic, and simple queries.
   - OLAP: Complex queries involving aggregations over large datasets.
5. **Backup:**
   - OLTP: Regular backups are crucial due to the critical nature of the transactional data.
   - OLAP: Backups might be less frequent, but the data recovery strategies can be complex due to the voluminous data.
6. **Indexes:**
   - OLTP: Uses primary keys, foreign keys, and indexes to speed up transactional processing.
   - OLAP: Uses bitmap indexes, materialized views, and partitioning for faster query processing.
7. **Data Update:**
   - OLTP: High frequency of write operations.
   - OLAP: Low frequency of write operations but high read operations. The data in OLAP systems is often periodically loaded from OLTP systems.
8. **Response Time:**
   - OLTP: Requires very fast response times (milliseconds) as it's usually customer-facing.
   - OLAP: Longer query response times (seconds to minutes or more) are acceptable as it's used for analytical purposes.
9. **Normalization:**
   - OLTP: Typically, the data is highly normalized.
   - OLAP: The data is denormalized, optimized for querying.
10. **Examples:**
    - OLTP: Point of sale systems, customer relationship management systems, online services.
    - OLAP: Data mining tools, reporting tools, business intelligence applications.

In a real-world scenario, businesses often integrate both OLTP and OLAP systems. OLTP systems support the day-to-day operations, while OLAP systems help with business analysis and decision-

making. Data from OLTP systems is often periodically extracted, transformed, and loaded (ETL process) into OLAP systems for analysis and the entire process flow between OLTP and OLAP is shown in Figure 5.6.

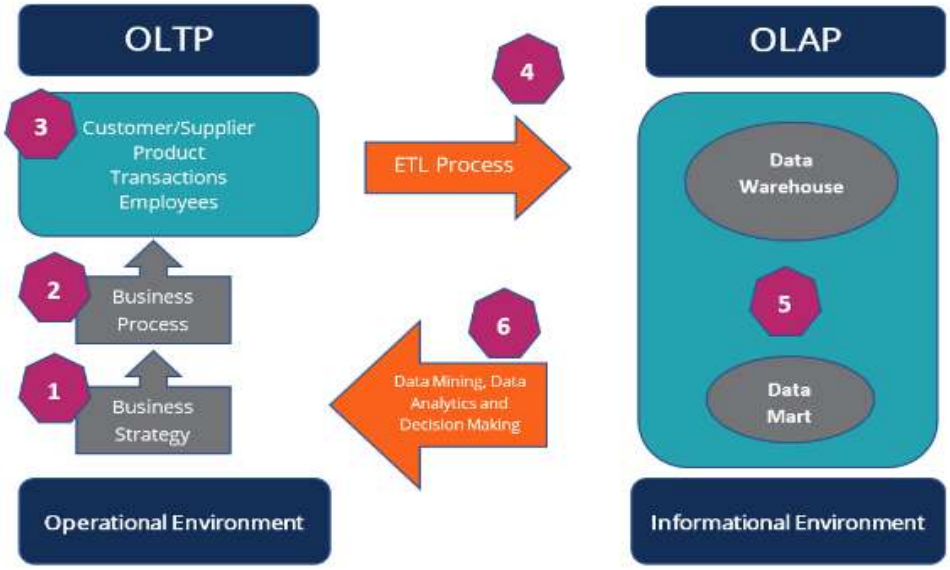


Figure 5.6 Process Flow Between OLTP and OLAP

5.5 Star Schema

The star schema is one of the most popular data modeling techniques used in data warehousing. Its structure is relatively simple, making it easy to understand and conducive for query performance. Here's a brief overview:

1. Central Fact Table:
   - At the heart of the star schema is the fact table. This table contains the quantitative data (often called "facts" or "measures") about specific events or transactions. Examples of facts include sales revenue, quantities sold, profit, etc.
   - The fact table usually has a composite primary key made up of foreign keys that link to associated dimension tables. This composite key helps in relating facts to their descriptive context.
2. Dimension Tables:
   - Surrounding the central fact table are several dimension tables. Each dimension table provides context for the data stored in the fact table.
   - Dimension tables typically contain descriptive, textual, or categorical information, often referred to as "attributes." These attributes give context to the quantitative data in the fact table.

- Examples of dimension tables could be: "Time" (with attributes like day, week, month, quarter, year), "Product" (with attributes like product name, category, manufacturer), "Customer" (with attributes like customer name, address, and phone number), and so forth.
- Each dimension table is linked to the fact table by a primary-to-foreign key relationship.

3. Characteristics:
   - Simplicity: One of the main advantages of the star schema is its simplicity. The clear distinction between fact and dimension tables makes it easy for end-users and developers to understand the database structure.
   - Performance: Due to its denormalized nature, the star schema is optimized for query performance. Queries often require fewer joins in a star schema than in more normalized structures like the snowflake schema.
   - Scalability: New dimensions or facts can be added without changing the existing structure, making the star schema flexible and scalable.
4. Drawback:
   - Redundancy: Because it's denormalized, the star schema can introduce data redundancy. This can lead to increased storage requirements and potential data integrity issues.
5. Usage:
   - The star schema is primarily used in OLAP systems, which are designed for complex queries and aggregations, rather than OLTP systems, which are transaction-oriented.

In graphical representations, the structure resembles a star, with the fact table in the center and dimension tables radiating outward, hence the name "star schema" as depicted in figure 5.7

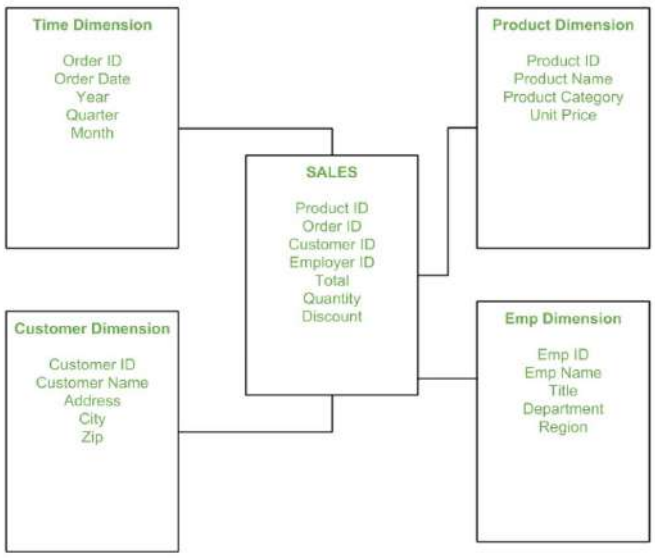


Figure 5.7 Star Schema

SALES is a fact table having attributes i.e. (Product ID, Order ID, Customer ID, Employer ID, Total, Quantity, Discount) which references to the dimension tables. Employee dimension table contains the attributes: Emp ID, Emp Name, Title, Department and Region. Product dimension

table contains the attributes: Product ID, Product Name, Product Category, Unit Price. Customer dimension table contains the attributes: Customer ID, Customer Name, Address, City, Zip. Time dimension table contains the attributes: Order ID, Order Date, Year, Quarter, Month.

5.6 Snowflake Schema

The snowflake schema is another common data warehousing model, closely related to the star schema. While both are used for OLAP (Online Analytical Processing), they have structural differences and a sample is shown in Figure 5.8. Here's an overview of the snowflake schema:

1. **Normalized Dimension Tables:**
   - In the snowflake schema, dimension tables are normalized. That means the data is organized within the database to reduce redundancy and improve data integrity. This is done by dividing the data into additional tables, creating a structure that looks like a snowflake, hence the name.
   - For instance, if you have a "Customer" dimension in a retail scenario, that dimension could be normalized into separate "Customer," "City," and "Country" tables instead of a single denormalized table containing all the information.
2. **Complex Structure:**
   - Because of this normalization, the snowflake schema tends to have a more complex structure than the star schema. Queries can become more complex and involve more table joins, potentially leading to longer query times.
3. **Reduced Data Redundancy:**
   - The main advantage of the snowflake schema is the reduction in data redundancy. This can lead to less storage space usage compared to the star schema.
   - However, the space saved may be minimal compared to the overall size of the data warehouse, and this saving might not justify the additional complexity.
4. En**hanced Data Integrity:**
   - The increase in normalization can improve data integrity, as the chances of inconsistent data are reduced. Any changes to a data point need to be made in just one place, reducing the risk of data anomalies.
5. **Query Performance:**
   - Query performance can be slower compared to the star schema due to the increased number of joins required by the normalization. However, modern databases are increasingly capable of mitigating this performance difference.
6. **Scalability Issues:**
   - While the snowflake schema can handle changing requirements by adding new dimensions easily, the complexity of the schema might increase significantly as the database scales, making maintenance more challenging.

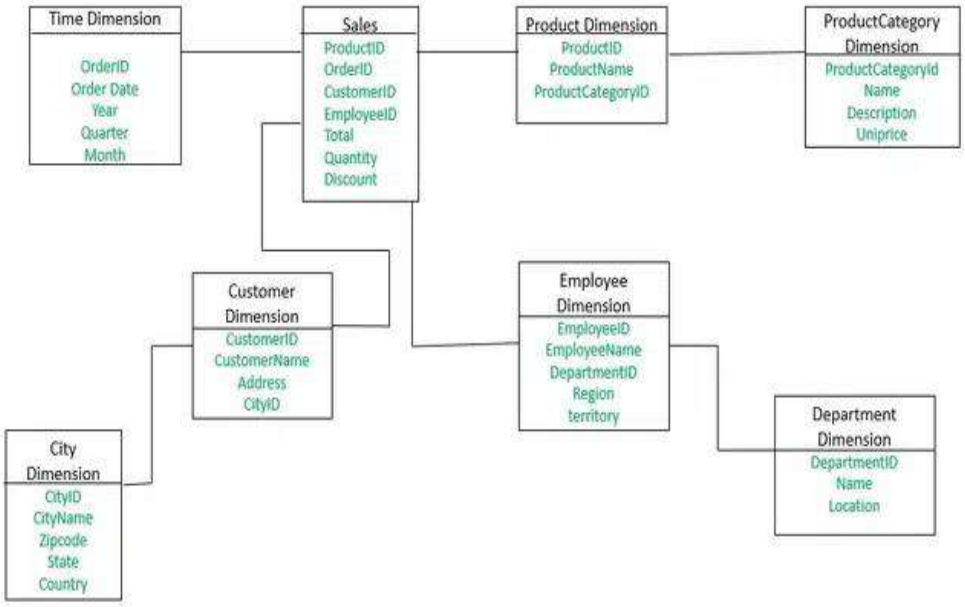


Figure 5.8 Snowflake schema

In practice, the choice between a star schema and a snowflake schema often depends on specific project requirements, the characteristics of the data being used, and the expected query performance. While the snowflake schema helps save storage space and ensures data integrity, it can increase complexity and affect performance. Conversely, the star schema is simpler and generally offers better query performance, but at the expense of greater storage space and potential data redundancy.

The Employee dimension table now contains the attributes: EmployeeID, EmployeeName, DepartmentID, Region, and Territory. The DepartmentID attribute links with the Employee table with the Department dimension table. The Department dimension is used to provide detail about each department, such as the Name and Location of the department. The Customer dimension table now contains the attributes: CustomerID, CustomerName, Address, and CityID. The CityID attributes link the Customer dimension table with the City dimension table. The City dimension table has details about each city such as city name, Zipcode, State, and Country.

5.7 Fact Constellation Schema

The Fact Constellation Schema, sometimes known as the Galaxy Schema, is an advancement over the star and snowflake schemas in data warehousing. Instead of having a single fact table, the fact constellation schema has multiple fact tables that share dimension tables. It essentially represents a complex multidimensional model and the general structure of the Fact Constellation schema is provided in Figure 5.9.

Here's an overview of the Fact Constellation Schema:

1. **Multiple Fact Tables:**
    - A fact constellation schema consists of multiple fact tables, each representing a different theme or subject. These fact tables are designed to answer different sets of business questions but can share some common dimensions.

- For example, a retail business might have separate fact tables for "Sales" and "Inventory." Both fact tables might share dimensions like "Time," "Product," and "Store," but each would also have unique measures specific to its subject.

2. **Shared Dimensions:**
   - In this schema, multiple fact tables can share the same dimension tables. This is useful when different business processes have overlapping descriptive attributes. The shared dimensions form the link between the different fact tables and can be used to analyze data across multiple subjects.
3. **Complexity:**
   - The fact constellation schema is more complex than the star or snowflake schemas due to the presence of multiple fact tables and the relationships between them.
   - It's necessary to carefully design and maintain the relationships between the tables to ensure data consistency and integrity.
4. **Analytical Power:**
   - The primary advantage of the fact constellation schema is its analytical power. By relating multiple fact tables through shared dimensions, users can perform comprehensive analyses across different business processes.
   - For instance, by relating "Sales" and "Inventory" fact tables through shared dimensions, one could analyze how sales trends impact inventory levels over time.
5. **Scalability:**
   - The schema is scalable in terms of adding new business themes or subjects. A new fact table with its unique measures can be introduced, and it can either have its own set of dimensions or share existing dimensions with other fact tables.
6. **Drawbacks:**
   - It can be challenging to design and maintain due to its complexity.
   - Query performance can be impacted if not designed optimally, especially when querying across multiple fact tables.

The fact constellation schema is particularly beneficial for large organizations with multiple business processes that have overlapping data dimensions. Properly designed, it can provide a holistic view of the organization's operations and offer deeper insights than more straightforward schemas. However, its complexity means it requires careful planning and a clear understanding of the business's analytical needs.

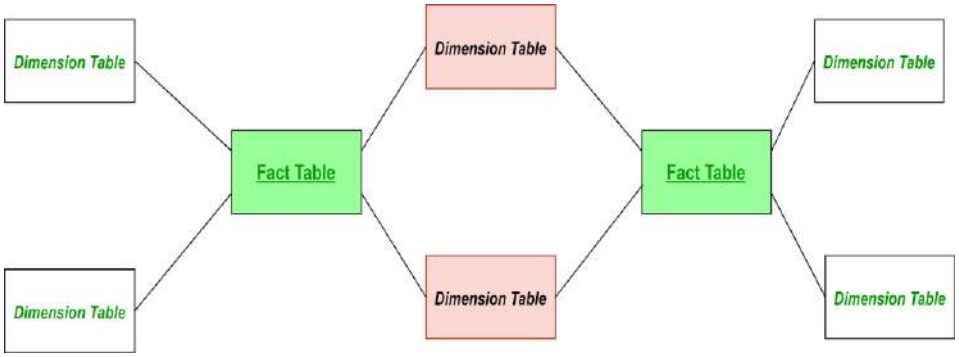


Figure 5.9 General structure of Fact Constellation

In the Figure 5.9, in both star schemas, the pink-colored Dimension tables are common to both, while the green-colored fact tables correspond to their respective star schemas.

Example: Figure 5.10

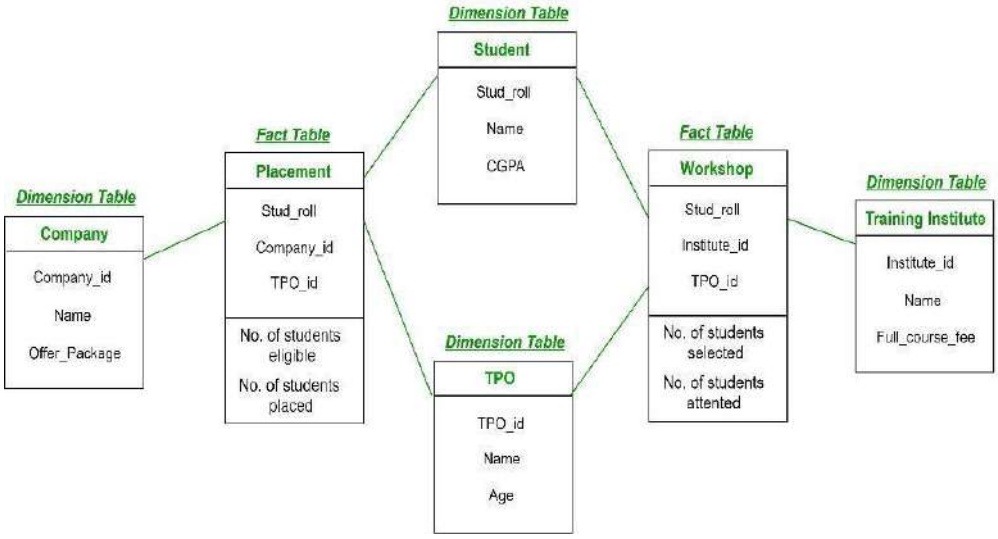


Figure 5.10 Example of Fact Constellation

In above demonstration:

- Placement is a fact table having attributes: (Stud_roll, Company_id, TPO_id) with facts: (Number of students eligible, Number of students placed).
- Workshop is a fact table having attributes: (Stud_roll, Institute_id, TPO_id) with facts: (Number of students selected, Number of students attended the workshop).
- Company is a dimension table having attributes: (Company_id, Name, Offer_package).
- Student is a dimension table having attributes: (Student_roll, Name, CGPA).
- TPO is a dimension table having attributes: (TPO_id, Name, Age).
- Training Institute is a dimension table having attributes: (Institute_id, Name, Full_course_fee).

So, there are two fact tables namely, Placement and Workshop which are part of two different star schemas having dimension tables – Company, Student and TPO in Star schema with fact table Placement and dimension tables – Training Institute, Student and TPO in Star schema with fact table Workshop. Both the star schema has two-dimension tables common and hence, forming a fact constellation or galaxy schema.

5.8 Data Mart

A data mart is a subset of a data warehouse, designed to support the specific needs of a particular business function or department, such as sales, marketing, finance, or HR. Essentially, it is a

scaled-down, more focused version of a data warehouse. Data marts cater to the specific analytical requirements of individual departments or business functions, simplifying users' interactions with data by narrowing the scope to what is most relevant to their specific domain. Data marts can be designed using star, snowflake, or fact constellation schemas, similar to a data warehouse. The choice of schema often depends on the specific analytical needs of the department. Data marts may source their data directly from various operational systems, but more commonly, they pull from an existing central data warehouse. Sourcing from a central data warehouse ensures consistent and harmonized data definitions across different departments. The Figure 5.11shows how the Data Marts are connected with Data Warehouse

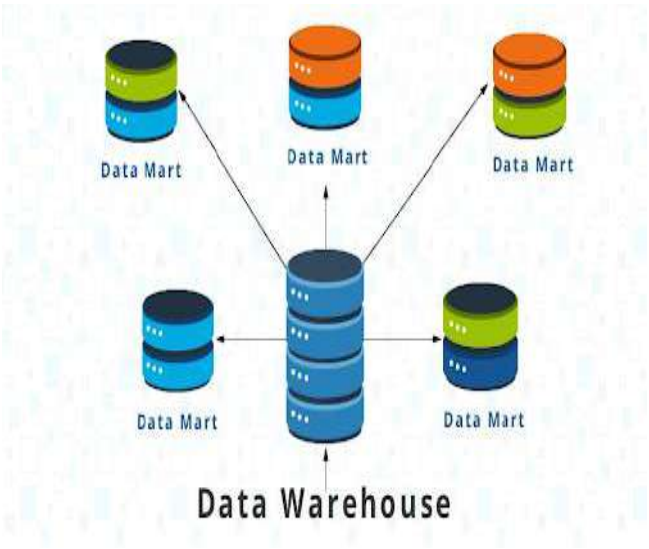
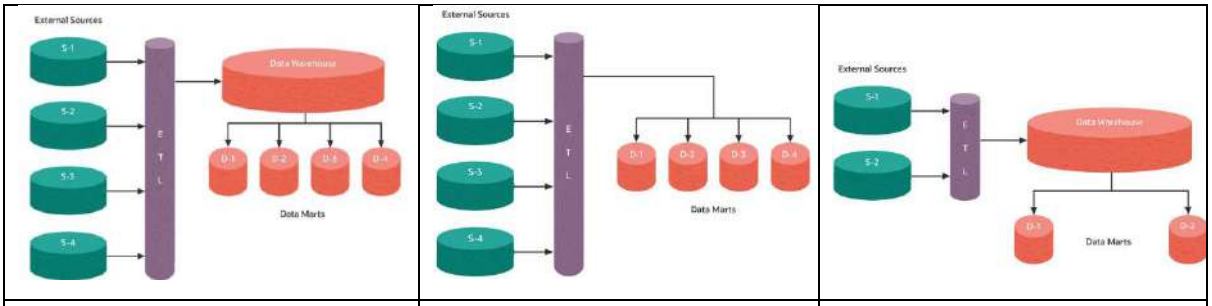


Figure 5.11 Data Mart

Types of Data Marts

Data marts can be differentiated in several ways, with a common method being based on their design approach or their connection to the primary data warehouse as depicted in Figure 5.12. A Dependent Data Mart originates from a central data warehouse, catering to a distinct department or organizational function. It sources its data from this central repository. Conversely, an Independent Data Mart is established without relying on a primary data warehouse. Instead, it usually pulls data from transactional systems or external data points. This method might be faster, but potential inconsistencies may arise, especially when multiple independent data marts exist in one organization. Meanwhile, a Hybrid Data Mart merges data from both the central data warehouse and outside sources, providing a mix of wide-ranging organizational data along with specific departmental or external insights. The selection among these variations is influenced by the organization's requirements, existing data infrastructure, budgetary limits, and implementation timeframe.



Dependent Data Mart	Independent Data Mart	Hybrid Data Mart

Figure 5.12 Types of Data Mart

In terms of performance, since data marts deal with a subset of data, queries tend to be faster compared to those in a large, comprehensive data warehouse. Regarding development speed, creating a data mart can be quicker than building a full-scale data warehouse due to its limited scope. This expedited process allows individual departments to gain analytical capabilities more rapidly. As for maintenance, data marts are generally easier to maintain than a complete data warehouse because of their smaller size and scope. However, if not properly managed, the existence of multiple data marts can lead to challenges in data consistency and redundancy.

5.9 Data Mining

Data mining is a computational process that uncovers patterns, correlations, trends, and anomalies within vast datasets using statistical methods, machine learning, and database systems. Its primary goal is to extract valuable information from these datasets and transform it into a structure that's easier to understand and use. The main objective of data mining is to identify patterns or insights from extensive data. Such insights can be applied in various domains, including market analysis, fraud detection, customer retention, production control, and scientific exploration

Key Features and Concepts of Data Mining:

1. Extraction of Information: Data mining aids in extracting meaningful patterns and insights from large volumes of data.
2. Machine Learning Integration: Many data mining techniques incorporate machine learning algorithms for pattern recognition and prediction.
3. Knowledge Discovery in Databases (KDD): This is the overall process of converting raw data into useful information. While data mining and KDD are often used interchangeably, data mining is actually one of the steps in the KDD process.
4. Association and Correlation: Discovering how events correlate or patterns that frequently occur together, such as in market basket analysis.
5. Clustering: Grouping similar data points together based on certain criteria, which can be helpful for segmentation.
6. Classification: Assigning predefined classes or labels to data points based on their attributes.

7. Anomaly Detection: Identifying unusual patterns that do not conform to expected behavior. It's often used in fraud detection.
8. Regression: Predicting a continuous value based on variable attributes.
9. Sequential Patterns: Discovering patterns in data sequences, often used in analyzing customer shopping sequences.
10. Visualization: Representing data in visual formats (charts, graphs, etc.) to identify patterns, trends, and anomalies.

Applications:

1. In finance, for fraud detection and credit scoring.
2. In marketing, for customer segmentation, market basket analysis, and targeted marketing.
3. In healthcare, for predicting illness outbreaks or patient diagnoses.
4. In retail, for inventory management and sales forecasting.
5. In manufacturing, for quality control and maintenance scheduling.

Challenges:

1. Dealing with noisy or incomplete data.
2. Ensuring privacy and security.
3. Managing the large volume and high dimensionality of data.
4. Overfitting, where a model may work well on the training data but not on new, unseen data.
5. Data mining tools and techniques have become essential in today's data-driven world, aiding businesses and researchers in making informed decisions.

5.9.1 Apriori Algorithm for Association Rule Mining

Association rule mining is a technique that shows how items are associated with each other. The classic example is the market basket analysis, where the goal is to discover relations between different items purchased together. For instance, when someone buys bread, they are likely to buy butter. The Apriori algorithm is based on the principle that a subset of a frequent itemset must also be a frequent itemset. For instance, if {apple, banana} is frequent, then {apple} and {banana} must also be frequent.

Steps:

1. Set a minimum support and confidence: Decide on a minimum value for support (how frequently an itemset appears in the dataset) and confidence (a measure of the probability that an item B is bought when item A is bought).
2. Generate frequent itemsets: Start with one-item itemsets and calculate their support by scanning the database. If the support is above the threshold, the itemset is deemed frequent.
3. Generate candidate itemsets: Increase the itemset size (e.g., from one-item to two-item itemsets). Generate these itemsets based on the previously found frequent itemsets (using the Apriori principle).

4. Prune the candidates: Remove those itemsets where some of the subsets aren't frequent. This step is based on the Apriori principle and helps in reducing the number of candidates checked in the next step.
5. Calculate the support for candidates: Scan the database and calculate the support for each of the remaining candidates.
6. Determine frequent itemsets: If the support for the itemset is above the threshold, keep it as a frequent itemset.
7. Generate association rules: For each frequent itemset, generate rules that have a confidence above the threshold. For instance, from the itemset {apple, banana}, one could generate the rule apple -> banana if the confidence is above the threshold.
8. Repeat: Continue the process until no more frequent itemsets can be generated.

Advantages:

1. Apriori is simple and easy to implement.
2. It's a level-wise search algorithm, meaning it explores one level of itemsets at a time.

Disadvantages:

1. It can be computationally expensive as it involves multiple scans of the database.
2. The performance can decrease with low minimum support values or large datasets.

The Apriori algorithm is a fundamental algorithm in association rule mining. Despite its limitations, particularly in terms of scalability, it has laid the groundwork for other algorithms and methodologies in the field.

Let’s understand the algorithm through the following example. The main two steps followed by this algorithm are Joining and Pruning.

Consider the following dataset for the example. Take minimum support = 2 and minimum confidence = 70%.

C1

Transaction ID	Item list
T1	A, B, E
T2	B, D
T3	B, C
T4	A, B, D
T5	A, B, C, E
T6	A, B, C

Dataset

L1

Item	Support Count
A	4
B	6
C	3
D	2
E	2

One-item Set

C2

Item set	Support Count
{A, B}	4
{A, C}	2
{A, D}	1
{A, E}	2
{B, C}	3
{B, D}	2
{B, E}	2
{C, D}	0
{C, E}	1
{D, E}	0

L2

Item set	Support Count
{A, B}	4
{A, C}	2
{A, E}	2
{B, C}	3
{B, D}	2
{B, E}	2

Two-items set frequent pattern

C3

Item set	Support Count
{A, B, C}	2
{A, B, E}	2

Three-items sets

Generate one-item set frequent pattern: Create a table that includes each item and corresponding support count of them (C1). Next, compare each candidate support count with a minimum support count (2). As all the support count values are not less than the minimum support count following will be the results (L1). In the first iteration of the algorithm, each item is a member of the set of candidates.

Generate two-items set frequent pattern: L1 join L1 to generate a candidate set of two item sets, C2. Then find out the support count for each item set. Check all subsets of an item set are frequent or not. If not frequent, remove that itemset. (Example subset of {A, B} are {A}, {B} they are frequent. Check for each itemset). After finding C2 compare each candidate itemset support to count with minimum support count (2). Remove the item sets that are less than the minimum support count.

Generate three-items set frequent pattern: For the generation of C3 compute L2 join L2.

C3 = {{A, B, C}, {A, B, E}, {A, C, E}, {B, C, D}, {B, C, E}, {B, D, E}}

Now the Join step is complete and then move on to the Prune step to reduce the size of the C3.

Apriori pruning principle: If there is any itemset which is infrequent, its superset should not be generated/tested!

Check if all subsets of these item sets are frequent or not. If not, then remove that itemset. (Here subset of {A, B, C} are {A, B}, {A, C}, {B, C} which are frequent. For {A, C, E}, subset {C, E} is not frequent so remove it. Similarly, check for every itemset)

So, the C3 will be as follows.

C3 = {A, B, C}, {A, B, E}

After comparing each candidate support count with minimum support count (2). As all the support count values are not less than the minimum support count L3 is the same as the C3.

Generate four-items set frequent pattern.

For the generation of C4 compute L3 join L3. After the Join step is complete move on to the Prune step to reduce the size of the C4.

Check all subsets of these item sets are frequent or not. Here itemset formed by joining L3 is {A, B, C, E}. It contains {A, C, E} which is not frequent. So, no itemset in C4.

The algorithm terminates as no frequent itemsets are found further.

It is time to generate strong association rules from the frequent items found. For that, we must calculate the confidence of each rule.

Confidence = Support_Count(Item set) / Support_Count(Subset)

Step 5: Generate the Association rule from frequent itemsets.

L = {{A}, {B}, {C}, {D}, {E}, {A, B}, {A, C}, {A, E}, {B, C}, {B, D}, {B, E}, {A, B, C}, {A, B, E}}

Let's take {A, B, E}

All non-empty subsets of selected frequent item set are,

{A}, {B}, {E}, {A, B}, {A, E}, {B, E}

As mentioned in the beginning, we consider the minimum confidence to be 70%.

R1 -> {A} ^ {B} => {E};

confidence = sup_c({A, B, E})/sup_c({A, B}) = 2/4 * 100 = 50%

R2 -> {A} ^ {E} => {B};

confidence = sup_c({A, B, E})/sup_c({A, E}) = 2/2 * 100 = 100%

R3 -> {B} ^ {E} => {A};

confidence = sup_c({A, B, E})/sup_c({B, E}) = 2/2 * 100 = 100%

R4 -> {A} => {B} ^ {E};

confidence = sup_c({A, B, E})/sup_c({A }) = 2/4 * 100 = 50%

R5 -> {B} => {A} ^ {E};

confidence = sup_c({A, B, E})/sup_c({B}) = 2/6 * 100 = 33%

R6 -> {E} => {A} ^ {B};

confidence = sup_c({A, B, E})/sup_c({E}) = 2/2 * 100 = 100%

So, the rules that are greater than the minimum confidence will be selected. Selected rules for {A, B, E} item set are R2, R3, R6.

Now you understand how the Apriori Algorithm works to find association rules hidden in a dataset.

5.9.2 Decision Tree Induction using Information Gain for Classification

Decision tree induction is a popular method used in machine learning for both classification and regression tasks. When using information gain for classification, the objective is to select attributes that yield the highest information gain for a decision. A decision tree models' decisions and decision-making processes in a tree-like structure, depicting choices and their potential consequences. Each node in the tree represents a feature (attribute), each branch represents a decision (rule) and each leaf represents an outcome (class label). A sample Decision Tree flowchart which has aforementioned features are shown in Figure 5.13.

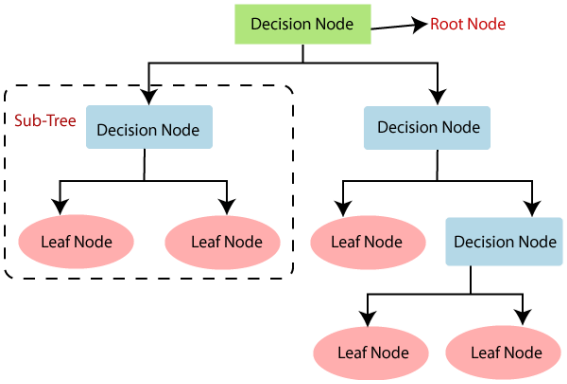


Figure 5.13 Decision Tree Classification Flowchart

Entropy is a measure of uncertainty or randomness. For a binary classification problem, the entropy of a set S with positive and negative examples is given by:

$$\text{Entropy}(S) = -p+\log_2(p+) - p-\log_2(p-)$$

Where:

- $p+$ is the proportion of positive examples in S
- $p-$ is the proportion of negative examples in S

Information gain is the decrease in entropy. IG indicates how much "information" a feature gives us about the class. It's calculated as:

$$IG(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} (|Sv| / |S|) \times \text{Entropy}(Sv))$$

Where:

- S is the current dataset for which entropy is being calculated.
- A is a feature or attribute.
- Values(A) is the set of all possible values for feature A.
- Sv is the subset of S for which feature A has value v.

Steps to Build Decision Tree using Information Gain:

1. **Initialization:** Start with the entire training set.
2. **Feature Selection:** For each attribute in the dataset, calculate the entropy and the information gain.
3. **Decision Making:** Select the attribute with the highest information gain as the decision node and split the dataset into subsets.
4. **Recursion:** Recursively build the tree by repeating the process for each subset until one of the stopping criteria is met, such as:
    - All tuples in a partition belong to the same class.
    - There are no remaining attributes.
    - No more instances.
5. **Tree Pruning (Optional):** To avoid overfitting, we can prune the tree by removing branches that have weak information gain.

Advantages:

- Simple to understand and visualize.
- Requires little data preprocessing.
- Suitable for both numerical and categorical attributes.

Disadvantages:

- Prone to overfitting especially when the tree is deep.
- Sensitive to small variations in the data.

- The highest information gain attributes are favored, which can lead to biased trees.

Example: Suppose you have a dataset that contains 4 features (outlook, temp, humidity, and windy) and one label (play).

- let's calculate its entropy, for the label value (It is discrete or categorical Yes/No), we have 9 yes and 5 no, so the entropy of our dataset is calculated as: 0.94
- Constructing a decision tree is all about finding an attribute that returns the highest Information Gain.
- Information Gain = Entropy(S) — [(weightedAvg) * Entropy(Each Feature)]
- let's calculate the Information gain for the feature "outlook"; "outlook" has three values Sunny (2yes/3no), Overcast(4yes/0no), and Rainy(3yes/2no).

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

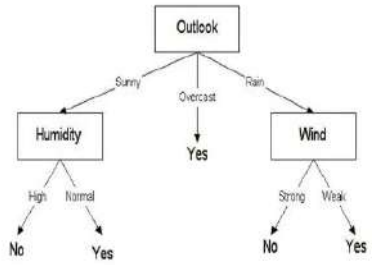
Dataset

$$E(S) = -P(Yes)\log_2 P(Yes) - P(No)\log_2 P(No)$$

$$E(S) = -(9/14)^*\log_2 9/14 - (5/14)^*\log_2 5/14$$

$$E(S) = 0.41+0.53 = 0.94$$

Entropy value for the dataset



Final Decision Tree

- Entropy(outlook = sunny) = -2/5 log2(2/5)-3/5log2(3/5) = 0.971
- Entropy(outlook = overcast) = -4/4 log2(4/4) = 0
- Entropy(outlook = rainy) = -3/5 log2(3/5)-2/5log2(2/5) = 0.971

The Weighted Avg is the weighting of each feature value by how many elements it has.

1. for outlook = sunny, weightedAvg = 5/14
2. for outlook = overcast, weightedAvg = 4/14
3. for outlook = rainy, weightedAvg = 5/14

Information Gain(outlook) = 0.94-[5/14 * E(sunny) + 4/14 * E(overcast) + 5/14 * E(rainy)

Information Gain(outlook) = 0.94-[5/14 * 0.971+ 4/14 * 0 + 5/14 * 0.971

Finally, we get Information Gain(outlook) = 0.247.

Using the same process the information gain for the other features (temp, humidity, and, windy).

Info Gain(outlook) = 0.247

Info Gain(temp) = 0.029

Info Gain(humidity) = 0.152

Info Gain(windy) = 0.048

What will be our root decision node? Outlook?

Exactly, right, we always take the feature with the highest Information Gain.

The next step is to find the next node in our decision tree. Now we will find one under sunny. We have to determine which of the following Temperature, Humidity or Wind has higher information gain.

Calculate parent entropy E(sunny)

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes

		play		
		yes	no	total
Temperature	hot	0	2	2
	cool	1	1	2
	mild	1	0	1
				5

	play	
Humidity	yes	no
high	0	3
normal	2	0

E(sunny) = -2/5 log2(2/5)-3/5log2(3/5) = 0.971

Now Calculate the information gain of Temperature. IG (sunny, Temperature)

E (sunny, Temperature) = (2/5) *E (0,2) + (2/5) *E (1,1) + (1/5) *E (1,0) =2/5=0.4

Now Calculate the information gain of Temperature.

IG (sunny, Temperature) =0.971 – 0.4 =0.571

IG (sunny, Humidity) = 0.971

IG (sunny, Windy) = 0.020

Here IG (sunny, Humidity) is the largest value. So Humidity is the node that comes under sunny.

Note: A branch with entropy more than 0 needs further splitting.

Decision trees with information gain are intuitive and easy-to-interpret models. They have been widely used for classification problems. However, in practice, algorithms like C4.5 (an

improvement over basic ID3 algorithm) that use a normalized version of information gain called 'gain ratio', or other methods like Gini impurity, might be preferred to combat some of the disadvantages of plain information gain.

5.9.3 K-Means Clustering

K-means clustering is a widely-used clustering algorithm that aims to partition a set of observations into a number of clusters (denoted as K), where each observation belongs to the cluster with the nearest mean. It's an iterative algorithm that tries to minimize the intra-cluster distances and maximize the inter-cluster distances.

Algorithm:

1. Initialization: Select K initial centroids, where K is the number of clusters you want. These centroids can be randomly selected from the data points or can be generated in other ways.
2. Assignment: Assign each data point to the nearest centroid. This step will create clusters.
3. Update: Recalculate the centroid of each cluster as the mean of all the data points in that cluster.
4. Repeat: Repeat the assignment and update steps until no changes in clusters or a maximum number of iterations is reached.

The objective of the K-means clustering algorithm is to minimize the sum of squared distances between the data points and their respective cluster centroids.

$$Objective = \sum_{i=1}^{K} \sum_{x \in c_i} ||x - \mu_i||^2$$

Where:

K is the number of clusters.

- C_i is the set of all points assigned to the i^{th} cluster.
- μ_i is the centroid of the i^{th} cluster.

Advantages:

1. Simple and easy to implement.
2. Efficient for large datasets.
3. Often produces tight clusters.

Disadvantages:

1. Assumes clusters to be spherical and equally sized, which may not be the case always.
2. The number of clusters K needs to be specified beforehand.
3. Sensitive to the initial placement of centroids. Multiple runs with different initializations can yield different results.

4. Can be trapped in local minima; hence, sometimes methods like K-means++ initialization or multiple initializations are used to get better outcomes.

Applications:

1. Market segmentation
2. Image compression
3. Document clustering
4. Anomaly detection

Example: Use the k-means algorithm and Euclidean distance to cluster the following 8 examples into 3 clusters: A1=(2,10), A2=(2,5), A3=(8,4), A4=(5,8), A5=(7,5), A6=(6,4), A7=(1,2), A8=(4,9).

The distance matrix based on the Euclidean distance is given below:

	A1	A2	A3	A4	A5	A6	A7	A8
A1	0	$\sqrt{25}$	$\sqrt{36}$	$\sqrt{13}$	$\sqrt{50}$	$\sqrt{52}$	$\sqrt{65}$	$\sqrt{5}$
A2		0	$\sqrt{37}$	$\sqrt{18}$	$\sqrt{25}$	$\sqrt{17}$	$\sqrt{10}$	$\sqrt{20}$
A3			0	$\sqrt{25}$	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{53}$	$\sqrt{41}$
A4				0	$\sqrt{13}$	$\sqrt{17}$	$\sqrt{52}$	$\sqrt{2}$
A5					0	$\sqrt{2}$	$\sqrt{45}$	$\sqrt{25}$
A6						0	$\sqrt{29}$	$\sqrt{29}$
A7							0	$\sqrt{58}$
A8								0

Suppose that the initial seeds (centers of each cluster) are A1, A4 and A7.

d(a,b) denotes the Eucledian distance between a and b. It is obtained directly from the distance matrix or calculated as follows: $d(a,b)=\text{sqrt}((x_b-x_a)^2+(y_b-y_a)^2))$
seed1=A1=(2,10), seed2=A4=(5,8), seed3=A7=(1,2)

epoch1 – start:

A1:
d(A1, seed1)=0 as A1 is seed1
d(A1, seed2)= $\sqrt{13}$ >0
d(A1, seed3)= $\sqrt{65}$ >0
➔A1 ∈ cluster1

A2:
d(A2,seed1)= $\sqrt{25}$ = 5
d(A2, seed2)= $\sqrt{18}$ = 4.24
d(A2, seed3)= $\sqrt{10}$ = 3.16 ← smaller
➔ A2 ∈ cluster3

A3:
d(A3, seed1)= $\sqrt{36}$ = 6
d(A3, seed2)= $\sqrt{25}$ = 5 ← smaller
d(A3, seed3)= $\sqrt{53}$ = 7.28
➔ A3 ∈ cluster2

A4:
d(A4, seed1)= $\sqrt{13}$
d(A4, seed2)=0 as A4 is seed2
d(A4, seed3)= $\sqrt{52}$ >0
➔ A4 ∈ cluster2

A5:
d(A5, seed1)= $\sqrt{50}$ = 7.07
d(A5, seed2)= $\sqrt{13}$ = 3.60 ← smaller
d(A5, seed3)= $\sqrt{45}$ = 6.70
➔ A5 ∈ cluster2

A6:
d(A6, seed1)= $\sqrt{52}$ = 7.21
d(A6, seed2)= $\sqrt{17}$ = 4.12 ← smaller
d(A6, seed3)= $\sqrt{29}$ = 5.38
➔ A6 ∈ cluster2

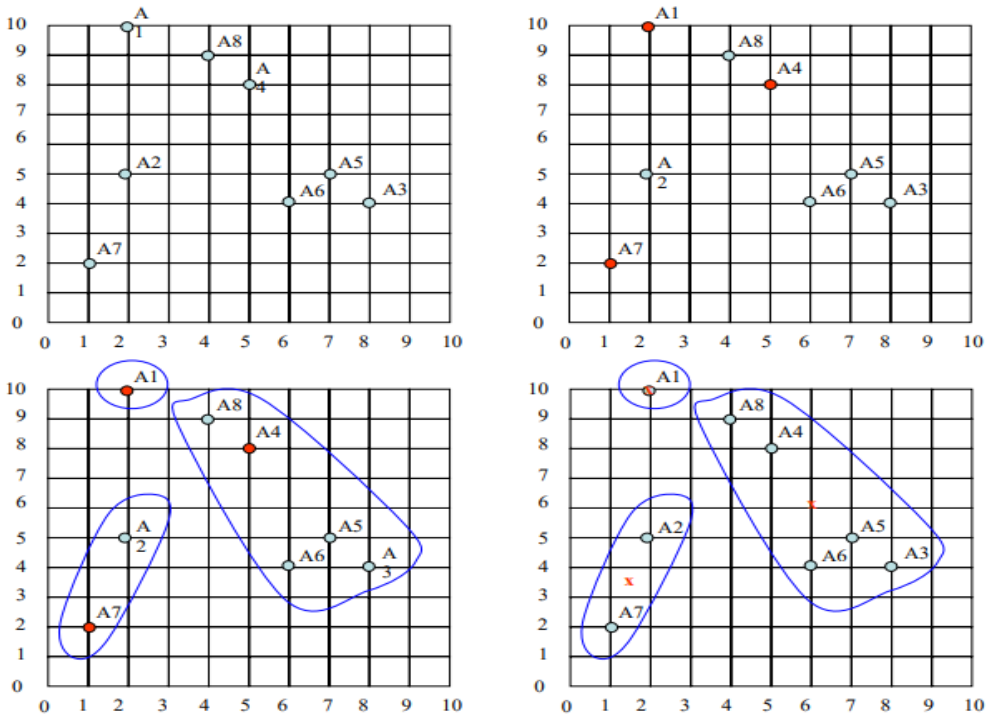
A7:
d(A7, seed1)= $\sqrt{65}$ >0
d(A7, seed2)= $\sqrt{52}$ >0
d(A7, seed3)=0 as A7 is seed3
➔ A7 ∈ cluster3

A8:
d(A8, seed1)= $\sqrt{5}$
d(A8, seed2)= $\sqrt{2}$ ← smaller
d(A8, seed3)= $\sqrt{58}$
➔ A8 ∈ cluster2

end of epoch1

new clusters: 1: {A1}, 2: {A3, A4, A5, A6, A8}, 3: {A2, A7}

centers of the new clusters:
C1= (2, 10), C2= ((8+5+7+6+4)/5, (4+8+5+4+9)/5) = (6, 6), C3= ((2+1)/2, (5+2)/2) = (1.5, 3.5)



We would need two more epochs. After the 2nd epoch the results would be:

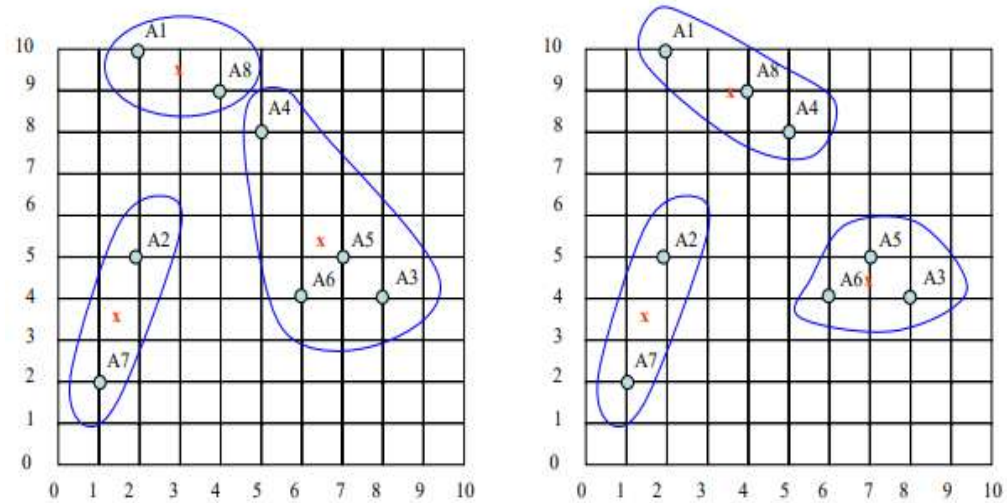
1: {A1, A8}, 2: {A3, A4, A5, A6}, 3: {A2, A7}

with centers C1=(3, 9.5), C2=(6.5, 5.25) and C3=(1.5, 3.5).

After the 3rd epoch, the results would be:

1: {A1, A4, A8}, 2: {A3, A5, A6}, 3: {A2, A7}

with centers C1=(3.66, 9), C2=(7, 4.33) and C3=(1.5, 3.5).



K-means clustering is a powerful algorithm but has its limitations. It's important to understand the underlying assumptions and the nature of the data before employing it. In practice, it might also

be helpful to use methods like the elbow method or silhouette analysis to determine an appropriate number of clusters (K).

5.10 Collaborative Learning Task

1. **OLAP vs. OLTP Debate**

   Instructions:

   1. Divide the class into two teams: Team OLAP and Team OLTP.
   2. Organize a debate where each team defends the benefits and use cases of their respective processing type.
   3. Conclude with a summary discussion on the primary differences and scenarios for using OLAP and OLTP.

2. **Hands-on Data Mining Workshop**

   1. Dataset Distribution: Provide each team with a sample dataset. It can be transactional data, customer behavior data, or any other relevant set.
   2. Implement Algorithms: Teams should implement:
      - Apriori Algorithm for Association Rule Mining
      - Decision Tree Induction using Information Gain for Classification
      - k-means Clustering
   3. Analysis: After applying these algorithms, teams should analyze the results, identifying patterns, associations, classifications, and clusters in their dataset.
   4. Share Findings: Each team should share their results with the entire group, explaining the patterns they've discovered and the potential implications or uses for these findings in real-world scenarios.

5.11 Self-Assessment questions and Exercises

Self-Assessment Questions:

1. Which of the following is NOT a characteristic of a Data Warehouse?

a) Real-time data processing

b) Consolidated data

c) Transformed data

d) Data storage for analytical purposes

2. In a three-tier architecture of a data warehouse, which layer is responsible for removing inconsistencies and transforming the data into a singular format?

a) Presentation Layer

b) Data Layer

c) Application Layer

3. Data cubes are primarily associated with which type of processing?

a) OLTP

b) OLAP

c) Batch Processing

d) Stream Processing

4. Which schema normalizes dimension tables to eliminate redundancy?

a) Star Schema

b) Snow Flake Schema

c) Fact Constellation Schema

5. A Data Mart is:

a) A comprehensive data storage system for the entire organization

b) A smaller, focused subset of a data warehouse for a specific business function

c) Another name for Data Warehouse

d) None of the above

6. The Apriori Algorithm is primarily used for:

a) Classification

b) Clustering

c) Regression

d) Association Rule Mining

7. Which technique uses information gain to determine the best attribute for splitting the data at a node?

a) Linear Regression

b) k-means Clustering

c) Decision Tree Induction

d) Apriori Algorithm

8. Which of the following is NOT a step in the k-means clustering algorithm?

a) Assigning each data point to the nearest cluster center

b) Calculating the mean of each cluster

c) Using information gain to update cluster centers

d) Iteratively adjusting cluster centers

Short Questions:

1. What is a Data Warehouse? List three of its main characteristics.
2. Briefly describe the Three Tier Architecture of a data warehouse.
3. What is a Data Cube, and why is it significant in the context of a data warehouse?
4. Differentiate between Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP).
5. In what scenarios would k-means clustering be particularly useful?
6. What is Association Rule Mining, and how does the Apriori Algorithm assist in it?

Exercise:

1. Design a Star Schema for a simple retail sales database, where the fact table contains Sale Amount, Quantity Sold, and Date of Sale, and the dimensions include Product, Customer, and Time.

2. Using any dataset of your choice, implement the Apriori algorithm to discover frequent item sets and derive association rules.

3. Take a sample dataset and apply the decision tree induction technique to classify the data based on certain attributes. Utilize the concept of information gain to make splits.

4. Given a set of data points, perform k-means clustering to categorize them into distinct groups. Analyze and interpret the clusters formed.

5.12 Case Studies

1. Transforming the Retail Landscape with a Data Warehouse

Background: A large retail chain with over 500 stores nationwide experienced difficulties in tracking sales, inventory, and customer behavior patterns. The existing system was fragmented, with data residing in various isolated systems.

Implementation: The company decided to set up a comprehensive data warehouse, adopting the three-tier architecture. They structured the data into a central data cube, facilitating faster OLAP operations as compared to their earlier OLTP system.

Design: Using the Star Schema, sales were placed at the center (fact table) surrounded by dimensions like Time, Product, and Customer. The design later evolved into a Snow Flake Schema to include detailed hierarchies like product categories and customer demographics.

Outcome: The new system allowed for faster, more in-depth analytics. Inventory management improved, and customer behavior patterns led to better marketing strategies. Moreover, individual stores, acting as data marts, could pull specific data relevant to their operations.

2. **Boosting E-commerce Sales with Data Mining Techniques**

Background: An emerging e-commerce platform, while witnessing steady growth, wanted to improve cross-selling and upselling on its platform.

Implementation: They used the Apriori Algorithm for Association Rule Mining to analyze customer purchase history, identifying frequently bought items together.

Design: Decision Trees, using information gain, were applied to classify customer types and predict future buying behavior based on past purchases and site interactions.

To further improve their marketing strategies, the k-means clustering method segmented their customer base into distinct groups, such as frequent buyers, occasional shoppers, and window shoppers.

Outcome: The platform saw a marked increase in sales from product recommendations. Targeted email campaigns to specific customer clusters also yielded higher click-through and conversion rates.

5.13 Summary

- **Data Warehousing:**
  - System where data is consolidated, transformed, and stored for analysis.
  - Characterized by three-tier architecture.
  - Uses data cubes for multidimensional analysis.
- **Analytical Processing Distinctions:**
  - OLAP (Online Analytical Processing):
    - Used for complex querying and reporting.
    - Utilizes schemas like Star Schema, Snow Flake Schema, and Fact Constellation Schema.
  - **OLTP (Online Transaction Processing):**
    - Used for day-to-day transactional processing.

- **Data Mart:**
  - A subset of a data warehouse.
  - Focused on a specific business area or function.
- **Data Mining Techniques:**
  - **Apriori Algorithm:** Used for association rule mining.
  - **Decision Trees:**
    - Popular for classification tasks.
    - Often induced using information gain.
- K-**means Clustering**
  - Used for segmenting datasets into groups or clusters.

Keywords

OLAP, OLTP, Clustering, Classification, Association Rule Mining, Decision tree, Data Mart.

5.14 Further readings

1. Data Mining-Concepts and Techniques -Jiawei Han & Michel Kamber. Morten Publisher 2nd Edition, 2006.
2. Data warehousing fundamentals for it professional, paulraj ponniah, john wiley & sons, inc., 2010.
3. The Elements of Statistical Learning: Data Mining, Inference, and Prediction" by Trevor Hastie, Robert Tibshirani, and Jerome Friedman
4. Building the Data Warehouse" by W. H. Inmon.