

DATABASE MANAGEMENT SYSTEM

Course Code: B24DS04DC
BSc Data Science and Analytics
Discipline Core Course
Self Learning Material



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Database Management System

Course Code: B24DS04DC

Semester - II

Discipline Core Course
Undergraduate Programme
BSc Data Science and Analytics
Self Learning Material
(With Model Question Paper Sets)



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala



DATABASE MANAGEMENT SYSTEM

Course Code: B24DS04DC

Semester- II

Discipline Core Course

BSc Data Science and Analytics

Academic Committee

Dr. Aji S.
Sreekanth M. S.
P. M. Ameera Mol
Dr. Vishnukumar S.
Shamly K.
Joseph Deril K.S.
Dr. Jeeva Jose
Dr. Bindu N.
Dr. Priya R.
Dr. Ajitha R.S.
Dr. Anil Kumar
N. Jayaraj

Development of the Content

Shamin S., Suramya Swamidas P.C.,
Rekha Raj C.T., Sumaja Sasidharan,
Greeshma P.P., Sreerekha V.K.,
Anjitha A.V., Aswathy V.S.,
Dr. Kanitha Divakar

Review and Edit

Dr. Aji S.

Linguistics

Dr. Aji S.

Scrutiny

Shamin S., Greeshma P.P., Anjitha A.V.,
Sreerekha V.K., Subi Priya Laxmi S.B.N.,
Aswathy V.S., Dr. Kanitha Divakar

Design Control

Azeem Babu T.A.

Cover Design

Jobin J.

Co-ordination

Director, MDDC :

Dr. I.G. Shibi

Asst. Director, MDDC :

Dr. Sajeekumar G.

Coordinator, Development:

Dr. Anfal M.

Coordinator, Distribution:

Dr. Sanitha K.K.



Scan this QR Code for reading the SLM
on a digital device.

Edition
May 2025

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-989642-9-8



All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

www.sgou.ac.m



Visit and Subscribe our Social Media Platforms

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

The University is committed to providing an engaging and dynamic educational environment that encourages active learning. The Study and Learning Material (SLM) is specifically designed to offer you a comprehensive and integrated learning experience, fostering a strong interest in exploring advancements in information technology (IT). The curriculum has been carefully structured to ensure a logical progression of topics, allowing you to develop a clear understanding of the evolution of the discipline. It is thoughtfully curated to equip you with the knowledge and skills to navigate current trends in IT, while fostering critical thinking and analytical capabilities. The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.



Regards,
Dr. Jagathy Raj V. P.

01-05-2025

Contents

Part - 1

Block 01	Introduction to DBMS and ER Model	1
Unit 1	Introduction to DBMS	2
Unit 2	Database System Architecture	15
Unit 3	ER Model	31
Unit 4	ER Model to Database Schema	52
Block 02	Structured Query Language	68
Unit 1	Functional Dependency	69
Unit 2	SQL Concepts	84
Unit 3	Built-in Functions	103
Unit 4	Views and Transaction Control Commands	114
Block 03	PL/SQL	127
Unit 1	Introduction to PL/SQL	128
Unit 2	Cursors	147
Unit 3	Procedures and Functions	159
Unit 4	Triggers	171

Part - 2

Block 04	Transaction Management	180
Unit 1	Introduction to Transaction Management	181
Unit 2	Serializability	192
Unit 3	Recovery	203
Unit 4	Deadlocks	215
Block 05	NoSQL Databases	226
Unit 1	Introduction to NoSQL	227
Unit 2	Types of NoSQL Databases	236
Unit 3	NoSQL Database Examples	249
Unit 4	NoSQL in Practice	288
Block 06	Distributed Database and Future trends	304
Unit 1	Distributed Database Concepts	305
Unit 2	Data Fragmentation and Replication	326
Unit 3	Big data and Data bases	341
Unit 4	Future Trends in Databases	351
Model Question Paper (SET 1)		363
Model Question Paper (SET 2)		366



Introduction to DBMS and ER Model



Unit 1

Introduction to DBMS

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ define the purpose of a Database Management System (DBMS)
- ◆ explain the advantages of using a DBMS over a traditional file system
- ◆ describe different types of data models used in DBMS
- ◆ identify the levels of abstraction in a database system

Prerequisites

In a hospital, patient records, doctor schedules, and billing details need to be interconnected to provide accurate and real-time updates. Without a centralized system, updating a patient's address in one file may not reflect in other records, leading to confusion and errors. Security is also a major concern, as sensitive data should only be accessible to authorized personnel while preventing unauthorized modifications. In modern applications, managing large volumes of data efficiently is crucial for ensuring accuracy, security, and quick access. Traditional file-based systems struggle with data redundancy, inconsistency, and slow retrieval, making them inefficient for handling complex operations. A structured approach is needed to organize, store, and retrieve information seamlessly while maintaining consistency across multiple users. Maintaining backups and ensuring data recovery in case of system failures is essential for business continuity. A well-designed system should also support multiple users accessing and updating information simultaneously without conflicts. Optimized data handling reduces storage wastage and enhances performance, making operations smoother and more reliable. Therefore, a structured data management solution is necessary for ensuring efficiency, security, and reliability in real-world applications.

Keywords

Entities, relationships, main memory, database administrator, Entity-Relationship model, data independence, data models.



Discussion

1.1.1 Getting Started with DBMS

Data needs to be stored in a fashion that it can be accessed very quickly and randomly when it is required and this is the primary purpose of a database. Database systems are used to manage large amounts of information. It requires both data definition structures for storage purpose and efficient mechanisms for its manipulation.

Organizations often need to manage multiple categories of data, which must be stored and retrieved in an organized and efficient manner to support various operations. A college requires data on components called as entities such as students, faculty, courses, and classes, as well as the relationships between these components. These relationships may include students enrolling in courses, faculty members teaching those courses, and classes being assigned for course sessions.

A database is an organized collection of related information that is structured in a way easy access, management and updating. In this context, a college database maintains comprehensive details about each component and their connections, ensuring data consistency and accessibility. A Database Management System (DBMS) enhances this process by providing a set of interrelated data and software programs designed to facilitate efficient access, manipulation, and management of the stored data. Through the use of a DBMS, organizations can maintain data integrity, support complex queries, and optimize their data operations for improved performance and decision-making.

1.1.2 Traditional File System versus DBMS

Traditional file system has some kinds of disadvantages as compared with database M.S. like more space, redundancy, lack of integrity, backup, etc. Some disadvantages of file system are

1. Storage issue

Need large space to store data so we need to use disk storage and for processing we need to bring the data to main memory. Required frequent data transfer from disk to main memory.

2. Redundancy

Data duplication occurs when repeated data is stored in multiple locations. Redundancy in file systems arises due to the lack of structured data management and relationships between files, resulting in multiple copies of the same data across different directories. Users may manually create duplicates, and file systems do not have built-in mechanisms to prevent this.

3. Difficulty in accessing related data

Difficulty in accessing related data occurs when information is stored across multiple files without a structured system. For example, in a file system, customer information may be stored in separate files for orders, invoices, and shipping records. If a user needs



to find all data related to a specific customer, they must manually search through each file to extract relevant details. This process is time-consuming and prone to errors, as there is no direct link between these files.

4. Limited security features

File systems have limited security features, offering only basic access restrictions such as read, write, or execute permissions. For example, a company storing employee data in a file system can only restrict access to entire folders, making it difficult to enforce role-based access to specific records. Implementing more advanced security, such as restricting access to sensitive fields like salary details, becomes complex and often requires manual workarounds. So only minimal access restrictions can be imposed and it will be complex to implement.

5. Data inconsistency

Data inconsistency occurs when multiple updates are made to the same data simultaneously without proper control. For example, in a file system, if two employees update a customer's contact information at the same time, one update may overwrite the other, leading to conflicting records. Concurrent access needs to be controlled; otherwise, multiple updates on the same data simultaneously will result in data inconsistency. Similarly, if any one updates in one file it will not effect in other file having some data, it cause data inconsistency.

1.1.3 Benefits of using DBMS

1. Data independence

Data independence refers to the ability to change the schema, which is the structure of data, without affecting other parts of the system. For example, if a database administrator adds a new column, such as "Date of Birth," to a table (internal schema), application programs that use the table will continue to function without needing any changes. Similarly, if the way data is stored on disk is modified (e.g., by adding an index to improve search speed), users and programs accessing the data are not impacted because the logical structure remains the same. This separation between internal storage and logical structure simplifies system updates and maintenance.

Data independence means ability to modify schema definition in one level without affecting the schema definition in the next higher level.

2. Efficient data access

Needed data can be retrieved in a convenient and efficient manner. Efficient data access allows users to retrieve the required data quickly and conveniently. In database systems, optimized indexing and query mechanisms help reduce search time. This improves performance and ensures faster access to large amounts of data without unnecessary delays.

3. Security and integrity of data

Security is maintained by allowing only authorized users to access the database,

preventing unauthorized access and data breaches. Integrity is ensured by managing concurrent access, meaning that when multiple users access or update the same data simultaneously, the system prevents conflicts and inconsistencies, ensuring that data remains accurate and consistent at all times. Only authorized users can access the database for maintaining security. Concurrent access to the same data will be managed for providing integrity.

4. Data administration

DBMS have central control of both data and programs accessing that data. The person having such Central control over the database system is called the database administrator (DBA).

5. Minimized Application development time

Minimized application development time refers to how a Database Management System (DBMS) reduces the effort needed to develop applications by managing many key functions like data storage, retrieval, and security. In traditional systems, developers had to create separate applications to handle tasks such as searching, updating, and maintaining data, which was time-consuming and complex. With a DBMS, these tasks are automated and standardized, allowing developers to focus on the core functionality of their applications, thus speeding up development and improving efficiency.

1.1.4 Data Storage using Data Models

A Database Management System (DBMS) allows users to define how data is stored using a **data model**, which serves as a set of high-level constructs to describe data and hides the complexities of low-level storage. This data model provides conceptual tools to represent data, relationships between data, the meaning (semantics) of the data, and rules to ensure data consistency. Among the various data models, the **relational model** is the most commonly used in modern DBMSs, where data is organized in tables with rows and columns.

A **semantic data model** offers a more abstract, high-level representation of data, making it easier for users to define their data structures initially. This abstract model can later be translated into a more concrete database design that the DBMS supports. One popular semantic data model is the **Entity-Relationship (ER) model**, which visually represents data through entities (objects) and their relationships, simplifying the design process for complex databases.

Relational Model

In the relational model, a relation is the primary data structure used to describe and organize data. A relation can be thought of as a table containing rows (records), where each record represents an individual entry, and columns (fields) define the attributes of that entry. The schema provides a description of the data in terms of the data model by defining the name of the relation (table), the names of its fields (attributes), and the data types for each field.

For example, in a college database that stores information about students, a relation

schema might look like this:

Student (*stud_id*: string, *stud_name*: string, *login*: string, *age*: integer, *gpa*: real).

This schema indicates that each record (row) in the Student relation will have five fields: a student ID, student name, login, age, and GPA. The data types such as string and integer specify the kind of data that each field can hold. This structured approach helps maintain consistency, allows efficient querying, and provides a clear framework for organizing and accessing data.

Table 1.1.1 Student Table

Stud_id	Stud_name	login	age	gpa
2100	Smith	Smith@cs	20	4.2
2155	Jack	Jack@hindi	21	3.9
2345	Nimal	Nimal@english	20	3.5
2454	Job	Job@history	22	3.8

From the above table 1.1.1, every row follows the schema of the Student relation. Each row in the Students relation is a record that describes a student. The description about a student is adequate for college purposes, so it can be considered as complete. Integrity constraints can be applied by specifying every student has a unique *stud_id*.

Relational data model used in various systems which includes IBM's DB2, Informix, Oracle, Sybase, Microsoft's Access, FoxBase, Paradox, Tandem and Teradata.

1.1.5 Alternative data models

Other important data models include

1. The hierarchical model

The hierarchical data model organizes data in a tree-like structure, where each piece of data, called a *record*, is connected by links that show relationships. In this model, one record (parent) can have multiple linked records (children), but each child can only have one parent. For example, imagine a university system where the main record is the University, which has several Colleges as children. Each College can have multiple Departments, and each Department can have many Students. For instance, the College of Applied Science might have a Computer Science Department, which includes students like "John Doe" and "Jane Smith." This structure makes it easy to quickly find related data, such as viewing all students in a department by following the links. However, it has some limitations, such as being difficult to update when new relationships are needed and being unable to easily handle complex relationships where data belongs to more than one parent. This model was commonly used in older systems like IBM's IMS for managing business data. The hierarchical model is used in IBM's IMS DBMS.

2. The network model

The network model represents data as a collection of records, with relationships between them defined by links that act like pointers. Unlike the hierarchical model, which follows a strict tree structure, the network model allows records to be connected in a more flexible structure called a graph. This means a record can have multiple parents and multiple children, making it suitable for complex relationships. For example, in a library database, a book record might be linked to multiple author records, while an author can be linked to multiple book records.

This model is commonly used in systems like IDS (Integrated Data Store) and IDMS (Integrated Database Management System). It provides faster data access for complex queries and is useful for applications requiring many-to-many relationships, such as supply chain networks and banking systems. However, managing and maintaining these links can be complex and requires careful planning.

3. The object oriented model

The object-oriented model organizes data as a collection of objects, similar to how objects are used in object-oriented programming. Each object represents a real-world entity and contains both data and methods. The data is stored in instance variables (attributes), and the methods define the operations that can be performed on the object.

For example, in a library database, a Book object might have attributes like "title," "author," and "ISBN," as well as methods like "borrow" or "return." This model makes it easier to handle complex data by encapsulating both data and behavior within objects. The object-oriented model is commonly used in advanced database systems such as ObjectStore and Versant, which are designed to efficiently manage large sets of interconnected objects. This approach simplifies the development of applications that require complex data structures, such as multimedia databases or simulations.

4. The object-relational model

The object-relational model is a combination of the relational model and object-oriented concepts. It enhances the relational model by allowing the database to store complex data types, such as objects, which are commonly used in object-oriented programming. In this model, data is still organized in tables like in the relational model, but those tables can contain objects with attributes (properties) and methods (functions). For example, consider a university database where a Student record can not only have basic fields like name and ID but also include an object for Address, which has its own fields like street, city, and zip code. This allows better data organization and reusability. Object-relational databases are used in many modern database systems, such as those from IBM, Oracle, and Informix. These systems support complex applications that need both structured data and advanced features like inheritance, encapsulation, and polymorphism, making data management more efficient and flexible.

1.1.6 Schema in DBMS

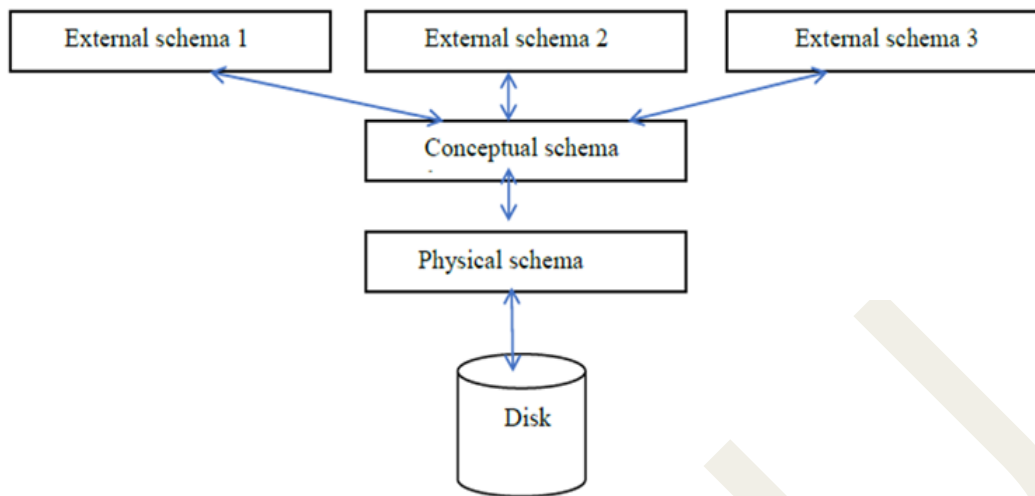


Fig. 1.1.1 Abstraction levels in database

Data in a DBMS describes at three levels of abstraction.

1. Physical Schema.
2. Conceptual Schema.
3. External Schema.

The Conceptual and External schemas in a database are defined using a Data Definition Language (DDL), which specifies the structure of the database. The Conceptual schema, also known as the logical schema, describes how data is organized and stored according to the data model of the Database Management System (DBMS). It defines the overall structure of the database by detailing all the tables (or relations) and their attributes. This schema provides a complete and consistent view of the data stored in the system without focusing on how the data is physically stored.

For example, in a university database, the conceptual schema might define the following tables:

- ♦ **Student:** Stores student details with attributes such as *stud_id*, *stud_name*, *login*, *age*, and *gpa*.
- ♦ **Faculty:** Contains information about faculty members, including *fac_id*, *fac_name*, and *sal*.
- ♦ **Course:** Represents academic courses with attributes like *c_id*, *c_name*, *credits*, and *enrolment*.
- ♦ **Classes:** Stores details of classrooms, including *class_no*, *class_address*, and *capacity*.
- ♦ **Enrolled:** Represents the relationship between students and courses, with attributes such as *stud_id*, *c_id*, and *grade*.

This conceptual schema helps in organizing and linking data logically and acts as a

blueprint for how data is represented and maintained across the system. The conceptual schema representation for above table is,

Student (stud_id: string, stud_name: string, login: string, age: integer, gpa : real)

faculty (fac_id: string, fac_name: string, sal: real)

course (c_id: string, c_name: string, credits: integer, enrolment: integer)

classes (class_no: string, class_address: string, capacity: integer)

Enrolled(stud_id: string, c_id: string, grade :string)

The physical schema explains how the data described in the conceptual schema is actually stored in the database. It specifies details like how the data is organized into files, how records are stored on disk, and whether indexes are created to speed up data retrieval. For example, a table containing student records may be stored as a collection of records or pages, and an index on the student ID may be created to allow faster searches.

Every database has one physical schema and one conceptual schema because there is only one physical storage system for the entire database. However, databases can have multiple external schemas, which define views, customized representations of the data for different users or purposes. These views do not store actual data but instead display data extracted from the database.

For example, if a student wants to see course details, a view can be created to show only the course name, faculty name, and number of enrolled students. The student does not need access to all other data in the system, such as administrative or financial records, which are hidden by the external schema. This structure improves both security and usability by tailoring data access to the needs of specific users.

Examples:

Courseinfo (c_id: string, fac_name: string, enrolment: integer)

As explained above Courseinfo is not included in the conceptual schema. It can be computed from the relations in the conceptual schema.

1.1.7 Data Independence

Data independence means that changes made to the database's internal structure do not affect how users interact with the data through applications. It allows application programs to be insulated from changes in the physical and conceptual schemas. This independence is achieved through three levels of data abstraction: the physical schema, conceptual schema, and external schema.

1. Physical Schema

The physical schema describes how data is actually stored on disk. It includes details like file structures, storage formats, and indexes. For example, a database administrator may decide to change the way student records are stored by creating an index on student IDs to speed up search operations. This change improves performance but does not impact how users or applications access the student records

2. Conceptual Schema

The conceptual schema provides a high-level view of the data, focusing on the organization of tables and relationships. For instance, a Student table may be linked to a Courses table through an enrollment relationship. Users working with this structure only need to understand how the tables are related, not how the data is physically stored.

3. External Schema (or View)

The external schema defines how specific users or applications view the data. It creates views that present only the necessary data while hiding other details. For example, a student may have access to a view that shows course information (course name, instructor, and enrollment count) without seeing sensitive data like faculty salaries.

Types of Data Independence

1. Physical Data Independence:

Physical data independence insulates users and application programs from changes to the physical storage of data. If changes are made to organisation of data on disk (e.g., switching from a sequential file format to a partitioned structure), users will not be affected. The database management system (DBMS) handles these changes internally. This allows administrators to optimize performance without disrupting application functionality.

2. Logical Data Independence:

Logical data independence protects users from changes to the conceptual schema, such as modifications to the organization or structure of the database tables. For example, in a college database, the Faculty table might originally have fields like faculty ID, name, department, and salary. To improve security, the database administrator may split this table into two separate relations:

- ◆ Faculty_Public (fac_id, fac_name, dept)
- ◆ Faculty_Private (fac_id, sal)

Now, sensitive information like salaries is stored in a separate table, but a CourseInfo view can be redefined to pull data from both relations. Users querying this view still receive the same results as before, without knowing that the underlying structure has changed. This ability to modify the logical structure while maintaining data access consistency is an example of logical data independence.

Consider a college database with multiple users, including students, faculty, and administrators. Each user needs access to different types of data:

- ◆ Students: They may view course information, including course names, faculty names, and enrollment numbers.
- ◆ Faculty: They can access information about their assigned courses and

students enrolled in those courses.

- ◆ Administrators: They have access to detailed records, including faculty salaries and department budgets.

Now, let's say the database is restructured to improve data privacy. The Faculty table is split into Faculty_Public and Faculty_Private, where sensitive data like salaries is moved to the Faculty_Private table. However, the CourseInfo view is updated to combine both tables and still provide complete information to authorized users. This change in the schema does not require updates to application programs or user queries because the external schema (view) remains the same.

If the database administrator decides to optimize performance by reorganizing how course records are stored on disk, users remain unaffected. The conceptual and external schemas abstract away these low-level details, maintaining a consistent user experience.

Advantages of Data Independence

Data independence is a critical feature of modern databases, achieved through the separation of physical, conceptual, and external schemas. By insulating users from changes at lower levels, it enhances flexibility, security, consistency and maintainability, making databases more efficient and reliable for both administrators and end-users.

Recap

- ◆ A Database Management System (DBMS) is used to store, retrieve, and manage data efficiently.
- ◆ Databases help organizations store structured data, such as students, faculty, courses, and their relationships.
- ◆ A DBMS provides mechanisms for data manipulation, integrity, and security.
- ◆ The file system has limitations like redundancy, difficulty in accessing related data, and data inconsistency.
- ◆ Unlike file systems, a DBMS provides better security, ensuring controlled access to data.
- ◆ Data independence in DBMS allows changes in schema without affecting applications.
- ◆ DBMS allows efficient data access through indexing and optimized querying.
- ◆ It maintains data security by granting access only to authorized users.
- ◆ A database administrator (DBA) centrally manages both data and programs in a DBMS.

- ◆ DBMS reduces application development time by providing built-in data management features.
- ◆ The relational model is the most widely used DBMS model, organizing data in tables with rows and columns.
- ◆ Other data models include hierarchical, network, object-oriented, and object-relational models.
- ◆ The hierarchical model organizes data in a tree-like structure with parent-child relationships.
- ◆ The network model represents data using graphs, allowing multiple parent-child relationships.
- ◆ The object-oriented model stores data as objects with attributes and methods, similar to programming languages.
- ◆ A DBMS has three levels of abstraction: physical schema, conceptual schema, and external schema.
- ◆ Data independence ensures that changes in physical storage or database structure do not impact user applications.

Objective Type Questions

1. What is the primary purpose of a database?
2. Which system manages large amounts of information efficiently?
3. What is the structured collection of data called?
4. What is a major drawback of the file system related to data duplication?
5. What issue arises due to difficulty in accessing related data in a file system?
6. What security limitation exists in file systems?
7. What term describes conflicting records due to multiple updates?
8. What concept allows schema changes without affecting higher levels?
9. What feature of DBMS ensures quick retrieval of data?
10. Who has central control over database management?
11. What reduces the effort needed to develop applications using a DBMS?
12. What model organizes data into tables with rows and columns?
13. What model represents data as a tree-like structure?

14. Which data model allows records to have multiple parents and children?
15. What model is based on objects with attributes and methods?
16. What model combines relational and object-oriented concepts?
17. What schema defines how data is physically stored?
18. What schema provides an overall logical structure of the database?
19. What schema represents user-specific views of data?
20. What type of data independence insulates users from storage changes?

Answers to Objective Type Questions

1. Storage
2. DBMS
3. Database
4. Redundancy
5. Manual Search
6. Limited Access
7. Inconsistency
8. Data Independence
9. Indexing
10. DBA
11. Automation
12. Relational
13. Hierarchical
14. Network
15. Object-oriented
16. Object-relational
17. Physical
18. Conceptual
19. External
20. Physical

Assignments

1. Explain the concept of data independence in a database management system (DBMS) with examples of different schema levels.
2. Compare and contrast file systems and database management systems in terms of redundancy, data access, security, and consistency.
3. Describe the relational data model, including its components such as tables, rows, and columns, with a detailed example.
4. Discuss the key advantages of using a DBMS over traditional file systems, focusing on security, integrity, data access, and administration.
5. Explain the hierarchical and network data models, including their structure, real-world applications, and advantages and limitations.

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems (3rd ed.)*. McGraw-Hill.
2. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011)., *Database system concepts (6th ed.)*. McGraw-Hill.

Reference

1. <https://www.geeksforgeeks.org/dbms>
2. https://www.tutorialspoint.com/dbms/dbms_architecture.htm

Unit 2

Database System Architecture

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ define database system architecture
- ◆ identify the components of one-tier, two-tier, and three-tier architectures
- ◆ describe how database architecture helps in data management
- ◆ summarize the benefits of different database architectures

Prerequisites

The structure of a database management system (DBMS) is crucial for handling large volumes of data while ensuring security, integrity, and scalability. Let's explore how different architectural approaches can be applied to a real-world example, using the context of an online retail store, "ShopSmart."

In a simple architecture, the database and the application are hosted on the same machine. Everything, from managing customer orders to processing payments, is done locally on a single computer. This setup works well for small operations with limited users, as it keeps things straightforward and fast. However, as the business grows, it becomes clear that this approach has limitations in terms of scalability. If the number of users increases or if the system fails, the whole operation can be compromised. Additionally, the lack of separation between data and application makes it harder to maintain or secure.

In a more advanced setup, the system separates the application from the database, with the database hosted on a dedicated server. Customers access the application via a web interface, and the application communicates directly with the database server to retrieve or store data. This setup centralizes the data, making maintenance easier and more efficient. It's more capable of handling larger volumes of data and users than the simpler approach, but as the business continues to grow, the database server may begin to face performance bottlenecks due to the increasing load from users. At this stage, the system might still struggle with scalability and security if not carefully managed.

The most robust approach involves introducing an additional layer that acts as an



intermediary between the users and the database. In this setup, users interact with the web interface, which sends requests to an application server. The application server processes business logic, handles user authentication, and interacts with the database server to retrieve or update data. The database server is dedicated solely to data storage, and the application server handles all the processing and logic. This separation allows for better scalability, security, and flexibility. By adopting such a structured approach, E-shop(e-commerce platform) can efficiently scale its operations, ensuring that performance remains optimal even as user demands increase. The system's ability to securely manage sensitive data, maintain consistency across different levels of operations, and support future growth is vital for the smooth functioning of the business. This layered design supports not only reliability and efficiency but also seamless integration of new technologies as the business evolves.

Keywords

External schema, Conceptual Schema, Physical schema, multi tier architecture, single tier architecture, File server, Client computer, transaction manager, Concurrency Control.

Discussion

A DBMS is software designed to manage large volumes of data efficiently. It ensures quick data access, concurrent transaction handling, data integrity, security, and system recovery from failures. It allows multiple users and applications to interact with the database simultaneously while maintaining data accuracy and security.

Data in a DBMS is structured based on a data model, which defines how data is organized and accessed. The most popular model today is the relational model, where data is stored in tables with rows (records) and columns (attributes). For example, in the retail system, there might be separate tables for products, customers, and orders, all related through keys and relationships.

The different levels of data management offer data independence, meaning that changes in data storage or organization do not disrupt user views or application logic. This separation simplifies system maintenance and improves flexibility, allowing the system to scale as data grows.

A DBMS is therefore crucial for organizations that rely on complex data operations. By offering efficient data access, security, and scalability, it helps businesses like Amazon deliver a smooth and reliable user experience, even with millions of transactions occurring every day.

1.2.1 Architecture of Database

The architecture of a DBMS can be either single-tier or multi-tier. In a single-tier architecture, the database and applications run on the same system, which is suitable

for smaller setups. In contrast, multi-tier architecture divides the system into multiple layers or modules, such as presentation, application, and database layers. This n-tier architecture allows for better scalability, security, and maintainability. For example, in a three-tier architecture, the user interface (external level) is handled by a web application, the application logic (conceptual level) processes user requests, and the database server (physical level) manages data storage and retrieval.

These three levels of abstraction in a DBMS provide a flexible and efficient way to manage large amounts of data, ensuring that users can access data securely and reliably without being affected by changes to the underlying storage or data structures. This design makes DBMSs suitable for a wide range of applications, from small businesses to large enterprises.

1.2.1.1 One tier architecture

The **one-tier architecture** is a simple database architecture where all operations, including data storage, application logic, and presentation services, are handled on a single machine, often referred to as the file server. In this setup, the communication occurs between the **file server** and the client computers. There is no dedicated separation between application services and the database, meaning both the data management and user interface are tightly coupled on the same server.

The file server is responsible for performing all major tasks, including:

1. **User Interface:** Providing a way for users to interact with the data, such as through forms or graphical interfaces.
2. **Presentation Services:** Handling how data is displayed to the user.
3. **Application Logic:** Processing user requests, managing business logic, and performing data manipulation operations.
4. **Data Storage:** Storing and retrieving files or data from the server's storage system.

For example, in a small office setup, a **local database application** installed on a file server may store all employee records. When a client computer requests data, the file server performs all the processing, retrieves the data, and returns the result to the client. The client computer mainly displays the output without performing any complex operations.

This architecture is simple to implement and suitable for small-scale environments with limited users. However, it has several limitations, including scalability and performance issues. As all tasks are processed on the file server, it can become a bottleneck under heavy workloads or with multiple concurrent users. Additionally, since the client computers depend entirely on the file server for data access and processing, any failure on the server can disrupt the entire system.

One-tier architecture is efficient for standalone applications but may not be suitable for larger organizations requiring high availability, performance, and scalability. Shown as Fig. 1.2.1 One tier architecture.

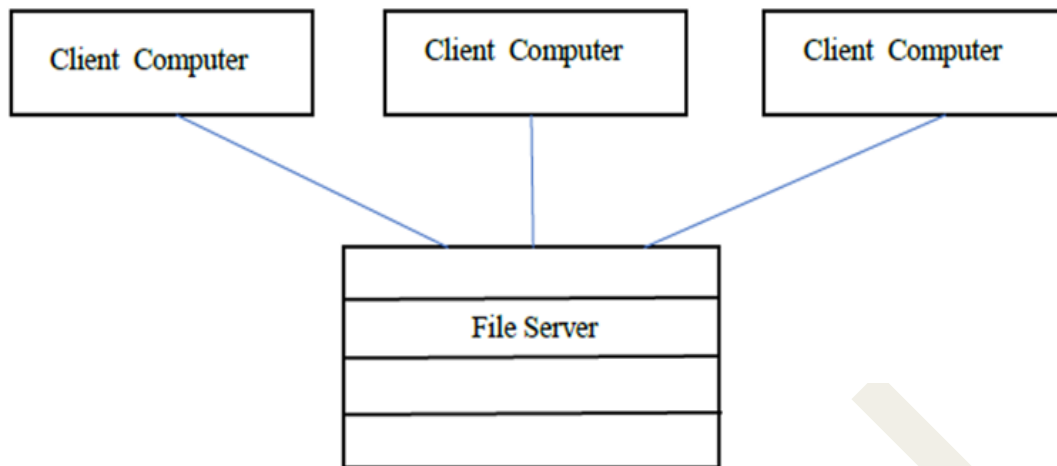


Fig. 1.2.1 One tier architecture

1.2.1.2 Two tier architecture

A **two-tier architecture** is a type of **client-server architecture** where communication occurs directly between client computers and the database server. In this setup, the client application (or user interface) and the database server are the two main components. The client sends requests to the server, such as queries to retrieve or update data, and the server processes these requests and returns the results to the client.

Structure of Two-Tier Architecture

Client Tier

This is the front-end application that users interact with. The client handles user inputs, displays data, and sends queries to the server. Examples include desktop applications like accounting software or student management systems.

Database Server Tier

The server is responsible for managing data storage, handling queries, and enforcing security and data integrity. It processes client requests, executes database operations, and sends responses back to the client. The server is typically located within the organization's network infrastructure.

How Communication Works

The client establishes a direct connection to the database server over the network. When a user performs an action on the client application (e.g., searching for a product in inventory), the application sends a query to the server. The server retrieves the requested data, processes it, and returns the result to the client, where it is displayed to the user. This real-time interaction enables quick and efficient data access.

Advantages of Two-Tier Architecture

1. Direct Communication

Clients communicate directly with the database server, reducing intermediate layers and improving response times.

2. Faster Performance

Since there are fewer components between the client and server, data retrieval and processing operations are faster compared to multi-tier architectures.

3. Simplified Setup

Two-tier systems are easier to develop and maintain due to their straightforward structure, making them suitable for smaller organizations or applications with limited users.

Consider a small college's student information system. In a two-tier setup, client computers used by faculty and staff run an application that connects directly to a database server within the college. When a faculty member queries student records, the client application sends a request to the server, retrieves the data, and displays it instantly. This setup allows for quick access to information without the need for additional layers of application processing. While two-tier architecture offers simplicity and speed, it may have limitations in scalability and security, especially when handling a large number of clients. In such cases, organizations may adopt a more complex multi-tier architecture to distribute workloads across different layers. Fig. 1.2.2 shows two tier architecture.

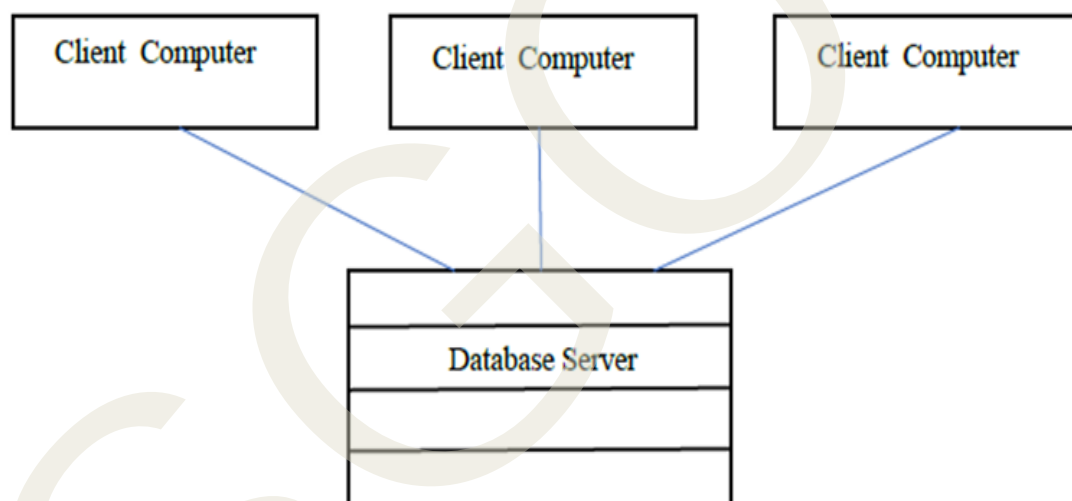


Fig. 1.2.2 Two tier architecture

1.2.1.3 Three tier architecture

The three-tier architecture is one of the most commonly used architectures in database systems, especially for web-based applications. It consists of three tiers: the client tier, business logic tier, and database tier. Each tier is independent of the others, with clear separation based on the complexity of user interactions and the way data is used and managed. This separation enhances security, scalability, and maintainability of the system.

1. Database Tier (Base Layer)

The database tier is the lowest layer, responsible for managing the actual storage and retrieval of data. This tier contains the database server and is accessed through a query

processing language such as SQL. It stores and organizes all data, enforces security, and manages transactions and data integrity.

In an e-commerce application, the database tier would store data related to products, customers, orders, and payments. The database server is unaware of the users directly accessing the system because all communication passes through the application tier.

2. Business Logic Tier (Middle Layer)

The business logic tier, also known as the application tier or middle tier, serves as a mediator between the client tier and the database tier. It processes business rules, logic, and operations required by the application. This tier handles complex tasks such as data validation, security checks, business rules enforcement, and interaction with external services. It presents an abstract view of the database to users by hiding the internal structure of data storage.

When a user places an order in an e-commerce application, the business logic tier verifies product availability, calculates totals, processes payment, and updates the database accordingly. This tier often runs on application servers and provides APIs or services that the client tier can interact with.

3. Client Tier (Presentation Layer)

The client tier is the topmost layer where users interact with the system. This tier is responsible for presenting data to users and collecting their input. It could be a web browser, mobile app, or desktop application. Users operate at this level without needing any knowledge of how the data is stored or processed internally. The client tier sends requests to the business logic tier, which then communicates with the database to fetch or manipulate data.

A user browsing products on an e-commerce website interacts with the client tier, which displays product details and allows the user to search, filter, and place orders.

How the Tiers Communicate

1. The client tier sends a request (e.g., search for a product) to the business logic tier.
2. The business logic tier processes the request, applies business rules, and interacts with the database tier to retrieve or update data.
3. The database tier performs the requested operations and sends the results back to the business logic tier, which formats the response and returns it to the client tier for display.

Advantages of Three-Tier Architecture

1. **Scalability:** Each tier can be scaled independently based on system load. Additional application servers can be added to handle more business logic requests without modifying the client or database tiers.

2. **Security:** The business logic tier acts as a gatekeeper, preventing unauthorized access to the database. Users only interact with the data through controlled interfaces.
3. **Maintainability:** Changes to one tier do not require changes to other tiers. The database structure can be modified without affecting the client interface.
4. **Reusability:** The business logic tier can serve multiple client applications, such as mobile apps, web browsers, and desktop applications, without duplicating code.

Example of Three-Tier Architecture in Action

In a university management system:

Client Tier: Students and faculty access the system through a web portal to view schedules, grades, and course details.

Business Logic Tier: The application server validates user credentials, applies access control, and processes data requests.

Database Tier: The database server stores all academic data, including student records, course information, and grades.

By dividing responsibilities across these tiers, the system becomes more organized, secure, and efficient. Fig. 1.2.3 shows three tier architecture.

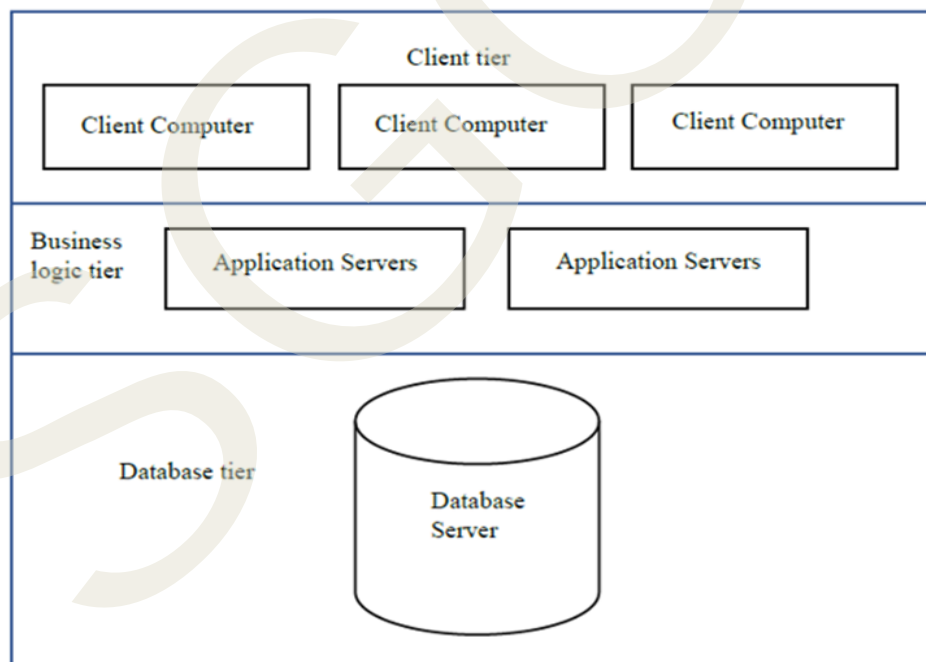


Fig. 1.2.3 Three tier architecture

1.2.2 Generalized DBMS Architecture

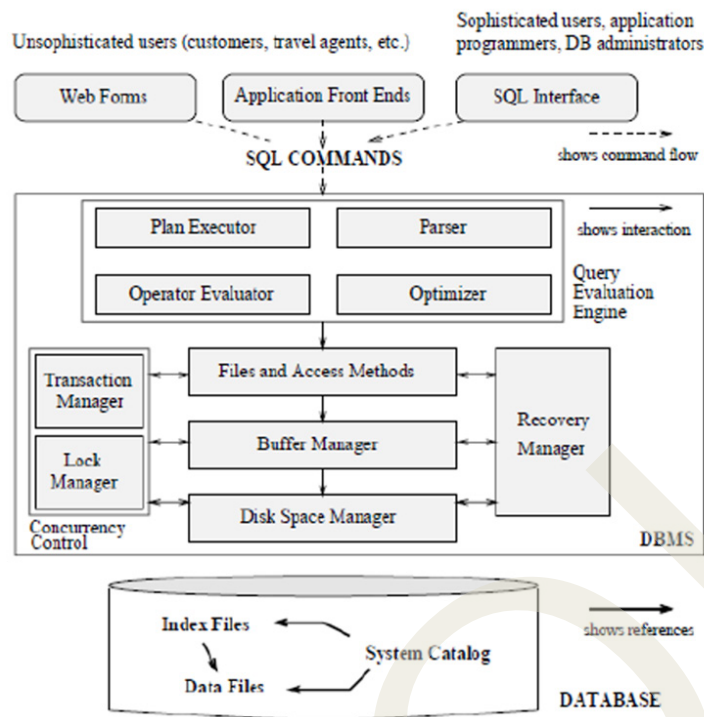


Fig. 1.2.4 General architecture of database management system

(Figure Source: Database Management Systems - Raghu Ramakrishnan and Johannes Gehrke, Third Edition, McGraw Hill, 2003)

1.2.2.1 Different types of DBMS users

A database management system caters to different categories of users, each with varying levels of technical expertise and roles in data access and management. These user types include unsophisticated users, sophisticated users, application programmers, and database administrators (DBAs).

1. Unsophisticated Users (End-Users)

Unsophisticated users are those who interact with the database through simple, user-friendly interfaces such as web forms, mobile apps, or application front ends. They do not require knowledge of the underlying database structure or how queries are executed. Their tasks often involve basic operations like searching, entering, or updating data.

In an e-commerce platform, customers browsing products or placing orders are unsophisticated users. They use the interface to view products without needing to know how the database stores or retrieves product information.

Key Characteristics

- ◆ Use pre-built applications or forms.
- ◆ Have no direct access to SQL or database queries.
- ◆ Focus on high-level tasks like data entry, search, and reporting.

2. Sophisticated Users

Sophisticated users have a deeper understanding of the database system and may interact with it directly using SQL or other query languages. These users typically perform complex data analysis, generate custom reports, or run ad-hoc queries to extract insights from the database.

A data analyst in a university may run SQL queries to generate reports on student performance or enrollment trends.

Key Characteristics

- ◆ Use direct query interfaces like SQL.
- ◆ Perform complex data retrieval and analysis.
- ◆ Require an understanding of database schemas and query optimization.

3. Application Programmers

Application programmers develop and maintain software applications that interact with the database. They write application code that includes SQL queries to retrieve, insert, update, or delete data. Programmers are responsible for implementing business logic and ensuring that applications communicate with the database effectively.

A programmer working on a banking application may write code to handle customer transactions, account updates, and loan processing.

Key Characteristics

- ◆ Develop applications that connect to the database.
- ◆ Write embedded SQL queries within application code.
- ◆ Focus on integrating business logic with database operations.

4. Database Administrators (DBAs)

Database administrators are responsible for the overall management, maintenance, and optimization of the database system. They ensure that the database operates efficiently, remains secure, and can handle concurrent user access without data conflicts or performance degradation. DBAs also perform tasks like backup and recovery, database tuning, user access management, and schema updates.

In a healthcare organization, a DBA might ensure that patient records are stored securely, that backups are performed regularly, and that users have appropriate access permissions.

Key Characteristics

- ◆ Manage database security, backups, and performance tuning.
- ◆ Control user access and maintain data integrity.
- ◆ Monitor system performance and optimize query execution..



1.2.2.2 Query Evaluation Engine

The Query Evaluation Engine is responsible for processing and executing SQL queries in a relational database system. It consists of several components that work together to generate query results efficiently.

1. Plan Executor

A query plan (or query execution plan) is a series of steps designed to retrieve data from the database. The query execution engine takes this plan as input, processes it, and returns the output for the user's query.

2. Parser

The parser breaks down a SQL statement into its components and organizes them into a data structure that other parts of the system can understand and process. This parsing stage is a critical step in query execution to ensure that the query is correctly interpreted.

3. Operator Evaluator

When a query uses operators like =, <, <=, >, >=, or ≠, the DBMS employs various techniques to evaluate these conditions efficiently.

The most commonly used methods for operator evaluation are:

1. Iteration

This method compares records by sequentially checking each entry in one table against another, similar to a nested loop join. It is simple but can be time-consuming for large datasets.

2. Indexing

If the columns used in the query condition have an index, the DBMS leverages the index to quickly locate matching records, avoiding full table scans. This significantly improves query performance by reducing the number of records that need to be checked.

3. Partitioning

In this method, the database divides a table into partitions or groups, often through techniques like hashing. When a query is executed, the DBMS searches within the relevant partition first, then scans only the records in that block, speeding up the search process.

Optimizer

The query optimizer is a crucial part of a Database Management System (DBMS) that evaluates Structured Query Language (SQL) queries to find the most efficient way to execute them. It creates one or more query plans, which outline possible methods for executing the query, and selects the plan that optimizes performance and resource usage.

1.2.2.3 Concurrency Control

Concurrency control in a DBMS is the process of managing multiple transactions that

occur at the same time to maintain atomicity, isolation, consistency, and serializability. When multiple transactions are executed simultaneously in a random order, it can lead to issues like data conflicts and inconsistency. Effective concurrency control is crucial to prevent these problems and requires timely management.

To implement concurrency control, a transaction manager and lock manager are used to coordinate and control access to shared data resources. These mechanisms ensure that transactions do not interfere with each other, preserving the accuracy and integrity of the database.

Transaction Manager

The transaction manager is a crucial component in a Database Management System (DBMS) responsible for coordinating transactions across one or more resources. It ensures that transactions are executed with atomicity (either all operations succeed or none are applied) and durability (completed transactions remain permanent even after a failure). The transaction manager creates and maintains transaction objects, monitors their progress, and ensures that all transaction properties are met.

In a banking system, a transaction to transfer money between accounts involves updating both the sender's and receiver's balances. The transaction manager ensures that either both updates occur successfully or neither of them is applied, preserving data consistency.

Lock Manager

The lock manager is responsible for implementing concurrency control by managing locking protocols. When multiple transactions attempt to access the same data simultaneously, the lock manager ensures that data access conflicts are handled correctly to prevent issues like dirty reads, lost updates, or deadlocks.

The lock manager maintains a lock table, which is a data structure used to track locks on data items. The lock table stores information about which transactions hold or are waiting for locks on particular data items. Fig. 1.2.4 shows the general architecture of the database management system.

Structure of the Lock Table

1. The lock table is implemented as a hash table, where the names of data items act as hash keys.
2. Each locked data item has a linked list associated with it.
3. Each node in the linked list represents a transaction that has requested a lock on that data item.
4. When a new lock request is made, a node is added to the end of the linked list.

If multiple transactions request a lock on a data item "Account_123," the lock manager creates a linked list for that item. The first transaction receives the lock, while subsequent requests are queued in the list. If no data item is locked, the lock table remains empty.

Consider a scenario where two transactions try to update the same customer's account balance at the same time. The lock manager ensures that one transaction acquires the lock first, completes its operation, and releases the lock before the other transaction can proceed. This prevents conflicts and ensures data consistency.

A DBMS includes several other components to manage different aspects of data handling: File manager, Buffer Manager, Disc Space manager and Recovery Manager are included.

1.2.2.4 File Manager, Buffer Manager, and Recovery Manager in DBMS

In a Database Management System (DBMS), several components work together to ensure efficient data storage, access, and protection. Among these components are the file manager, buffer manager, and recovery manager, each with distinct roles and responsibilities.

File Manager

The file manager is responsible for managing the hierarchical structure of files and folders within the database system. It provides users and applications with essential functions to organize and manipulate files and directories. These functions include creating, copying, moving, renaming, and deleting folders and files. However, while the file manager can create and manage folders, applications are responsible for creating individual files.

For example, in a university database system, the file manager may organize student data files into folders for different departments (e.g., "Engineering" or "Science"). Within these folders, applications can create files such as transcripts or enrollment records.

Buffer Manager

The buffer manager handles the allocation of memory buffers to optimize data access between main memory and disk storage. When a user or application requests data, the buffer manager checks if the data block is already in the buffer. If the block is unavailable, the buffer manager allocates space to load the block from disk.

To maintain efficient buffer usage, the buffer manager may need to free up space by removing older or less frequently used blocks. This process is known as buffer replacement and typically follows algorithms such as LRU (Least Recently Used) or FIFO (First In, First Out).

In an e-commerce system, the buffer manager may temporarily store product details and user search results in memory, allowing faster access for frequently requested data. If the buffer is full, older search results might be removed to make space for new data.

Recovery Manager

The recovery manager plays a crucial role in ensuring that the database can recover from failures such as hardware crashes, power outages, or data corruption. It performs both backup and recovery operations to protect the database's integrity. The recovery

manager stores metadata about these operations in the control file of the target database, which helps track backup history and recovery points.

The database administrator (DBA) has key responsibilities in managing the recovery process, including:

1. **Managing Complexity:** The DBA must plan and implement efficient backup and recovery strategies.
2. **Scalability and Reliability:** Backup processes should handle large amounts of data without failures.
3. **Hardware Utilization:** The DBA ensures optimal use of hardware resources (e.g., storage devices) during backups.
4. **Efficient Recovery:** Recovery time should be proportional to the amount of data being restored, minimizing downtime.

In a banking system, the recovery manager can restore transaction logs to ensure that customer account balances remain accurate after a system crash. Regular backups and transaction logs help reduce data loss and allow for quick recovery.

These managers work together to maintain a stable and efficient database environment. The file manager organizes data storage, the buffer manager optimizes data access, and the recovery manager safeguards data integrity by handling backup and recovery operations.

Recap

- ◆ Database architecture defines how data is stored, accessed, and managed in a system.
- ◆ A good database system avoids redundancy and ensures data consistency.
- ◆ It also protects sensitive data through security and access controls.
- ◆ Database systems should be scalable to handle more users and data efficiently.
- ◆ There are two main types of DBMS architecture: single-tier and multi-tier.
- ◆ Single-tier architecture runs the database and application on the same machine.
- ◆ Multi-tier architecture separates system functions into layers like presentation, application, and database.
- ◆ The three levels of abstraction in DBMS are external, conceptual, and internal levels.
- ◆ These levels help in managing data independently of how it's stored.
- ◆ One-tier architecture combines data storage, logic, and interface in a single system.

- ◆ It is suitable for small systems with limited users and low data load.
- ◆ A major downside of one-tier architecture is poor scalability and performance under load.
- ◆ Two-tier architecture separates the client application and the database server.
- ◆ Clients send queries directly to the server, which processes and returns results.
- ◆ This setup is faster and simpler but may face issues with large numbers of users.
- ◆ Three-tier architecture has client, business logic, and database layers.
- ◆ The client tier handles user interaction through apps or web browsers.
- ◆ The business logic tier processes rules, validation, and interacts with the database.
- ◆ The database tier stores, manages, and secures the data.
- ◆ Three-tier systems offer better scalability, security, and maintenance.

Objective Type Questions

1. What type of architecture runs the application and database on the same machine?
2. Which architecture divides the system into presentation, application, and database layers?
3. In three-tier architecture, what is the middle tier called?
4. What kind of architecture is most suitable for small-scale environments with limited users?
5. In two-tier architecture, what component directly communicates with the database server?
6. Which tier in three-tier architecture stores and retrieves data?
7. In three-tier architecture, which layer handles user interaction?
8. What is the main drawback of one-tier architecture?
9. Which user type interacts through forms or graphical interfaces without using SQL?
10. Who is responsible for managing and optimizing database systems?
11. What is the role of the middle tier in three-tier architecture?

12. Which architecture offers better scalability and maintainability—single-tier or multi-tier?
13. What do we call users who write application code containing SQL?
14. Which architecture allows a web browser to interact with a back-end database via an application server?
15. What term describes users who generate reports using SQL?
16. In two-tier architecture, what tier processes the user's SQL queries?
17. Which architecture isolates the user from the internal structure of the database?
18. Which user type needs knowledge of database schemas and query optimization?
19. What is the lowest layer in three-tier architecture?
20. What kind of system can become a bottleneck due to lack of load distribution?

Answers to Objective Type Questions

1. One-tier
2. Three-tier
3. Application tier
4. Two-tier architecture
5. Client
6. Database
7. Client
8. Scalability
9. Naive user
10. DBA
11. Processing
12. Multi-tier
13. Application Programmers
14. Three-tier

15. Sophisticated
16. Client tier
17. Three-tier
18. Sophisticated
19. Database
20. One-tier

Assignments

1. Compare and contrast one-tier, two-tier, and three-tier database architectures in terms of structure, communication flow, advantages, and limitations. Provide relevant examples for each architecture.
2. Discuss the roles of the file manager, buffer manager, and recovery manager in a DBMS. Explain how these components work together to maintain data integrity and optimize system performance.
3. Describe the concept of concurrency control in a DBMS. Explain the importance of ensuring atomicity, isolation, consistency, and serializability in concurrent transactions, along with the roles of the transaction manager and lock manager.
4. Explain the query processing steps in a relational database system. Describe the roles of the parser, optimizer, plan executor, and operator evaluator, and how they contribute to efficient query execution.

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems* (3rd ed.). McGraw-Hill.
2. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). *Database system concepts* (6th ed.). McGraw-Hill.

Reference

1. <https://www.geeksforgeeks.org/dbms>
2. https://www.tutorialspoint.com/dbms/dbms_architecture.htm

Unit 3

ER Model

Learning Outcomes

Upon the completion of the unit, the learner will be able to:

- explain the Entity-Relationship (ER) Model
- famialiarize Symbols and Components in ER Diagrams
- explain the Concept of Keys in DBMS
- state the purpose of an ER diagram in database design

Prerequisites

Think about how you organize information in your daily life—like saving contacts in your phone. Each contact has a name, phone number, and email, which help identify and retrieve the right person when needed. Similarly, databases store and organize information in a structured way to make retrieval and management efficient.

In your previous learning, you have come across the basic idea of data and how it is stored. Now, to store data systematically in a database, we use Entity-Relationship (ER) models. The ER model helps in designing databases by representing real-world objects (entities) and their connections (relationships).

In this unit, you will explore different types of attributes, such as simple, composite, and derived attributes, which define the properties of an entity. You will also learn about keys, which uniquely identify records in a database, and relationships, which show how entities are connected. Understanding these concepts will give you a strong foundation for designing well-structured databases that can handle complex real-world applications efficiently.

Keywords

Semantic data model, Entity-Relationship model, attributes Candidate key, Composite key, super key, primary key, weak entity



Discussion

Designing a database for an application typically involves several key steps. First, we gather requirements—both functional and data-related—by consulting with database users to understand their needs. Next, we create a logical or conceptual design, where the Entity-Relationship (ER) model plays a crucial role as the most widely used graphical representation of a database's conceptual structure. Finally, we move to the physical database design, which includes aspects like indexing for performance optimization, and external design, such as defining views to control data access and presentation.

A high-level model in DBMS is a conceptual framework that represents data in an abstract and user-friendly way, independent of physical storage details. It focuses on how data is organized and related, making it easier for users to understand and design databases. The Entity-Relationship (ER) model is a common high-level model, using entities, attributes, and relationships to depict real-world data. This model helps in the early stages of database design, ensuring a clear structure before moving to the logical and physical levels of implementation.

1.3.1 Entity-Relationship (ER) model

The Entity-Relationship (ER) model is a high-level conceptual framework used in database design to visually represent the structure of data and its relationships. Proposed by Peter Chen in 1976, the ER model helps designers and developers understand and organize data before implementing it in a database. It consists of entities (real-world objects like students, employees, or products), attributes (characteristics of entities like name, age, or ID), and relationships (connections between entities like "enrolled in" or "works for"). The model is typically represented using ER diagrams, which use symbols such as rectangles for entities, ovals for attributes, and diamonds for relationships. The ER model plays a crucial role in designing relational databases by ensuring data is properly structured, reducing redundancy, and enhancing efficiency.

1.3.1.1 Importance of ER diagrams in Database Designs

- ◆ ER diagrams represent the E-R model in a database, making them easy to convert into relations (tables).
- ◆ ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.
- ◆ ER diagrams require no technical knowledge of the underlying DBMS used.
- ◆ It gives a standard solution for visualizing the data logically.

1.3.1.2 Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols it is shown in fig 1.3.1

- ◆ Rectangles: Rectangles represent Entities in the ER Model.

- ◆ Ellipses: Ellipses represent Attributes in the ER Model.
- ◆ Diamond: Diamonds represent Relationships among Entities.
- ◆ Lines: Lines represent attributes to entities and entity sets with other relationship types.
- ◆ Double Ellipse: Double Ellipses represent Multi-Valued Attributes.
- ◆ Double Rectangle: Double Rectangle represents a Weak Entity.
- ◆ Double Diamonds → Represent Weak Relationships.








Symbol	Representation	Description
	Entity	Represents a real-world object or concept.
	Attribute	Defines properties of an entity.
	Primary Key	Uniquely identifies an entity.
	Derived Attribute	Computed from other attributes.
	Multivalued Attribute	Can have multiple values for a single entity.
	Relationship	Represents a relationship between two or more entities
	Weak Entity	Represents an entity that cannot be uniquely identified by its own attributes and requires a strong entity for identification.

Fig. 1.3.1 Symbols in ER model

1.3.2 Components of ER Diagram

ER diagram used to show the relationship among entity sets. ER Diagram is able to represent the complete logical structure of a database. ER Diagram can be easily converted to schema.

ER model includes show in fig 1.3.2:

1. Entity and entity set.
2. Attributes and type of attributes.
3. Keys

4. Relationships among entities.

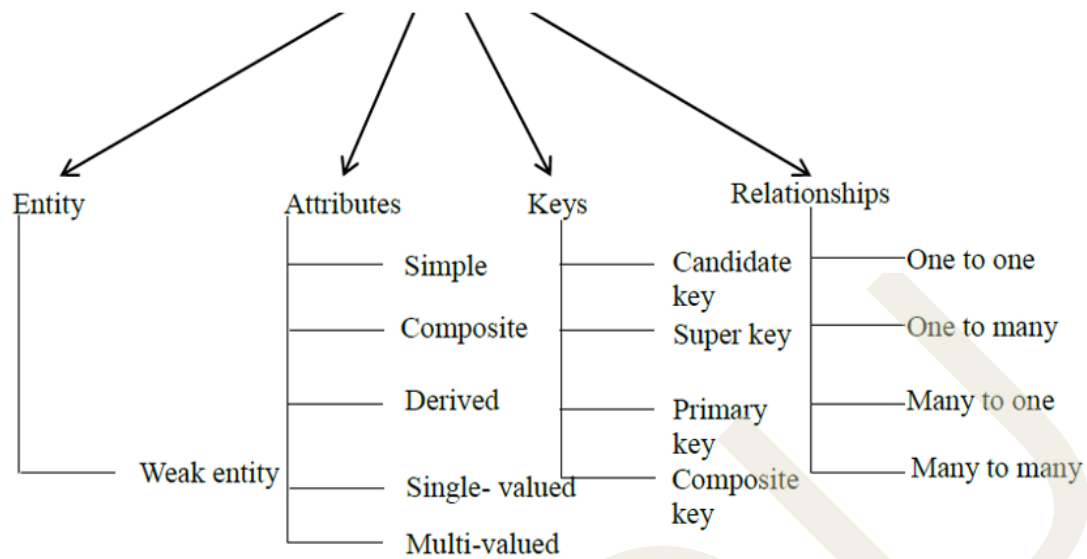


Fig. 1.3.2 Components of ER Model

1.3.2.1 Entity and Entity Set

An entity represents a real-world objects that have a distinct identity. They can be physical (e.g., a student, an employee, a car) or conceptual (e.g., a job position, an order).

An Entity is an object of Entity Type and a set of all entities is called an entity set. For Example, E1 is an entity having Entity Type Student and the set of all students is called Entity Set. In ER diagram, Entity Type is represented as in fig 1.3.3 :

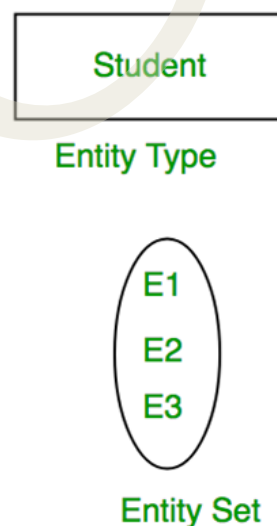


Fig. 1.3.3 Entity set

We can represent the entity set in ER Diagram but can't represent entity in ER Diagram because entity is row and column in the relation and ER Diagram is graphical representation of data.

Types of Entities

- ♦ **Strong Entity:** An entity that has a **primary key** and can exist independently. Student is a strong entity because it can exist independently.
- ♦ **Weak Entity:** An entity that **depends** on a strong entity and does not have a primary key of its own. It relies on a **foreign key** (from the related strong entity) and has a **partial key** (discriminator) to differentiate its instances. Dependent (child of an employee) is a weak entity because it exists only if the Employee entity exists.

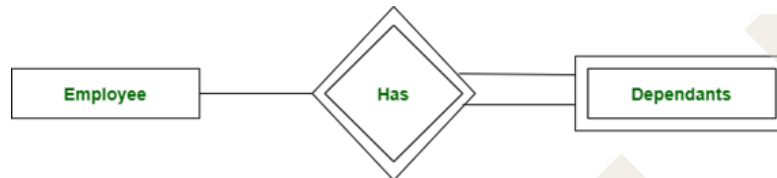


Fig. 1.3.4 Strong and weak entity

1.3.2.2 Attributes in the ER Model

In the Entity-Relationship (ER) Model, attributes define the characteristics or properties of an entity or relationship. Attributes provide meaningful information about entities and help in uniquely identifying them. Fig 1.3.5 shows attribute symbol,



Fig. 1.3.5 Attribute symbol

Types of Attributes in the ER Model

a. Simple attribute

Simple attributes are that have atomic values, which cannot be split up further. An example for it is `stud_id` in student entity. Fig 1.3.6 shows example of Simple attribute

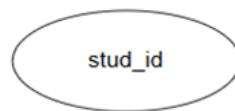


Fig. 1.3.6 Simple attribute

b. Composite attribute

Composite attributes have non-atomic values made up of more than one simple attribute. Consider address as an example, it contains `house_name`, `city_name`, `pincode` etc.

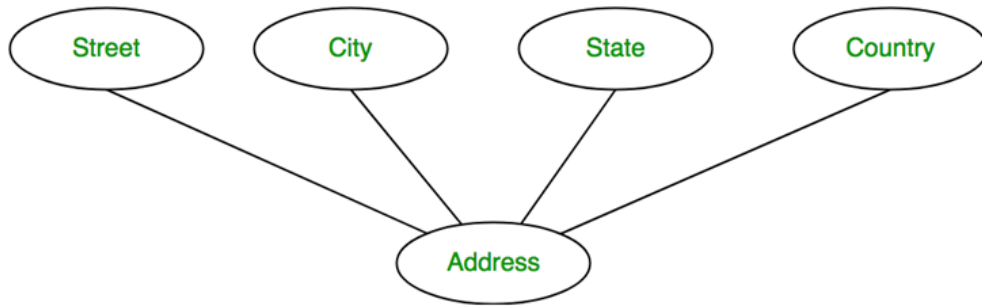


Fig. 1.3.7 Composite attribute

c. Derived attribute

Derived attributes are not directly present in the database management system; it can be derived using other attributes. For example age is a derived attribute which can be computed from date_of_birth attribute it is shown in fig 1.3.8.

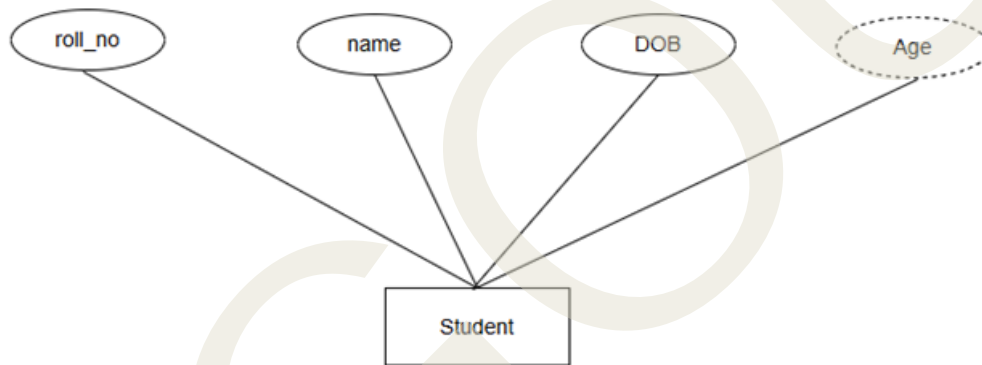


Fig. 1.3.8 Derived attribute

d. Single-valued attribute

Single-valued attributes have single value, example stud_id in student entity.

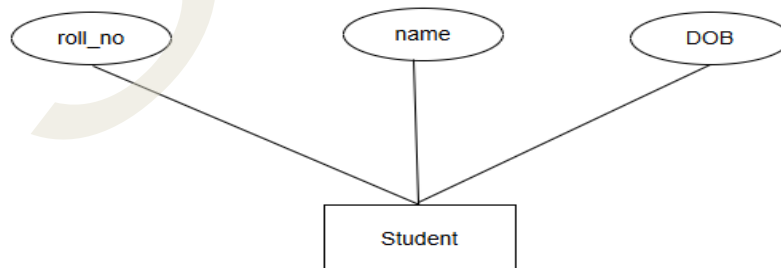


Fig. 1.3.9 Single valued attribute

e. Multi-valued attribute

Multi-valued attributes that can hold multiple values. For example phone_number is a multi-valued attribute because a student can have multiple phone numbers. It is represented in the ER Diagram using double ovals.



Fig. 1.3.10 Multivalued attribute

The Complete Entity Type Student with its Attributes can be represented as:

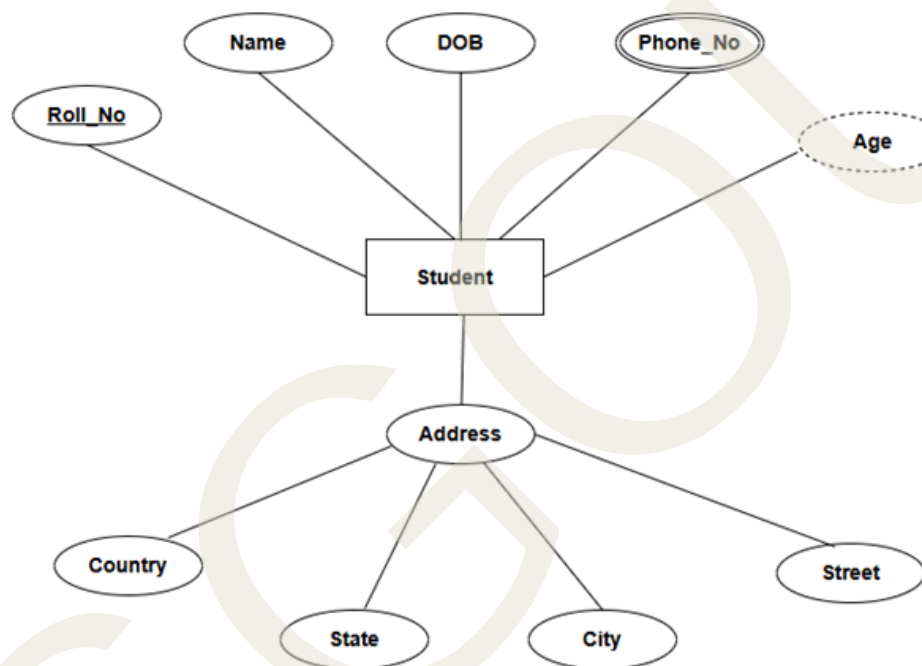


Fig. 1.3.11 Multi valued attribute

1.3.2.3 Keys in ER Model

In a database, keys are attributes or sets of attributes that help uniquely identify records in a table and establish relationships between tables. We require keys in a DBMS to ensure that data is organized, accurate, and easily accessible. Keys help to uniquely identify records in a table, which prevents duplication and ensures data integrity. Keys also establish relationships between different tables, allowing for efficient querying and management of data. Without keys, it would be difficult to retrieve or update specific records, and the database could become inconsistent or unreliable.

Different keys are:

- ◆ Candidate key

- ◆ Super key
- ◆ Primary key
- ◆ Composite key
- ◆ Foreign Key

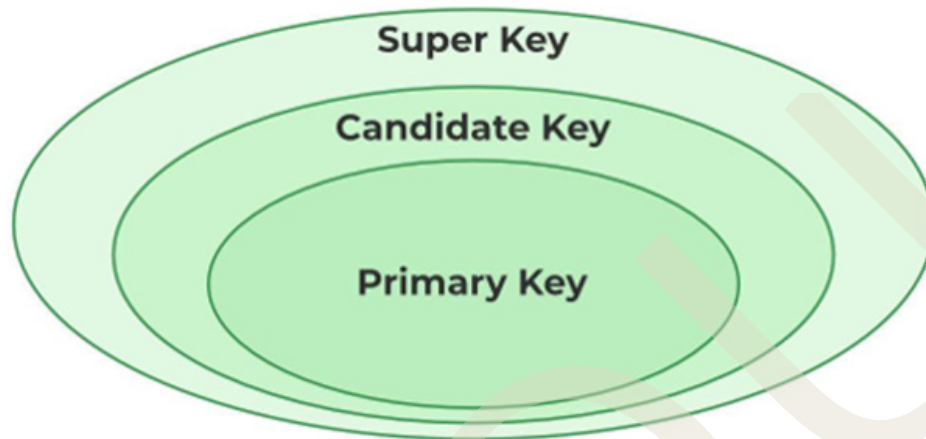


Fig. 1.3.12 Keys hierarchy in ER model

Consider an example of a university entity with attributes `univ_id` and `univ_name`, both `univ_id` and `univ_name` can act as a Candidate key for the table as they contain unique and non-null values.

a. Candidate key

The minimal set of attributes that can uniquely identify a tuple is known as a candidate key. A table can have multiple candidate keys.

<code>Stud_id</code>	<code>Sname</code>	<code>Address</code>	<code>Phno</code>
----------------------	--------------------	----------------------	-------------------

The student has three attributes: `Stud_id`, `Sname`, `Address` and `Phno`. Here `stud_id` can be considered as a candidate key. `stud_id` attributes can uniquely identify the tuples.

b. Super key

A **superset of a candidate key** that uniquely identifies records. It is a single key or a group of multiple keys that can uniquely identify tuples in a table. It supports NULL values in rows. A **candidate key** is a minimal super key, meaning it has no unnecessary attributes.

<code>Stud_id</code>	<code>Sname</code>	<code>Address</code>	<code>Phno</code>
----------------------	--------------------	----------------------	-------------------

For example (`Stud_id`, `Sname`), (`Stud_id`, `Address`), (`Stud_id`, `Phno`), (`Stud_id`, `Sname`, `Address`), (`Stud_id`, `Sname`, `Phno`), (`Stud_id`, `Address`, `Phno`), (`Stud_id`, `Sname`, `Address`, `Phno`) can be considered as super keys as they all uniquely identify the tuples of the table. Because of the presence of the `stud_id` attributes which are able to uniquely identify the tuples. The other attributes in the keys are unnecessary, but as a combination it can uniquely identify tuples.

c. Primary key

There can be more than one candidate key in relation out of which one can be chosen as the primary key that uniquely identifies records. This candidate key is called the primary key. A unique identifier for each record in a table. Cannot have duplicate or NULL values.

Stud_id	Sname	Address	Phno
---------	-------	---------	------

Here we can choose Studid as primary key, it can identify each tuple uniquely.

d. Composite key

In some cases, there is no single attribute that can uniquely identify each tuple. In such cases, more than one attribute is taken as the primary key; this is called a composite key.

Sname	Address	Phno
-------	---------	------

In this example, Sname alone cannot identify each tuple. A combination of any two or all three attributes can uniquely identify each row.

e. Alternate Key

An alternate key is any candidate key in a table that is not chosen as the primary key. In other words, all the keys that are not selected as the primary key are considered alternate keys. An alternate key is also referred to as a secondary key because it can uniquely identify records in a table, just like the primary key.

Stud_id	Sname	Address	Phno
---------	-------	---------	------

Here Sname, Address and Phno is an alternate key.

f. Foreign Key

A foreign key is an attribute in one table that refers to the primary key in another table. The table that contains the foreign key is called the referencing table, and the table that is referenced is called the referenced table.

Student Table

Stud_id	Sname	Address	Phno
---------	-------	---------	------

Course Table

Stud_id	Teacher's Name	Course Name
---------	----------------	-------------

Stud_id is a primary key of the table Student, and foreign key of the table Course. Here Course is a referencing table and Student is a referenced table.

1.3.2.4 Relationship

It is used for showing the relation between entities. Diamond is used to represent relationships in an ER Diagram. A relationship set can be thought of as a set of n-tuples: Each n-tuple denotes a relationship involving n entities e₁ through e_n, where entity e_i

is in the entity set E_i . For example in fig 1.3.13, 'Enrolled in' is a relationship type that exists between entity type Student and Course. In the ER diagram, the relationship type is represented by a diamond and connects the entities with lines.



Fig. 1.3.13 Relationship in ER model

A set of relationships of the same type is known as a relationship set. The following relationship set depicts S1 as enrolled in C2, S2 as enrolled in C1, and S3 as registered in C3.

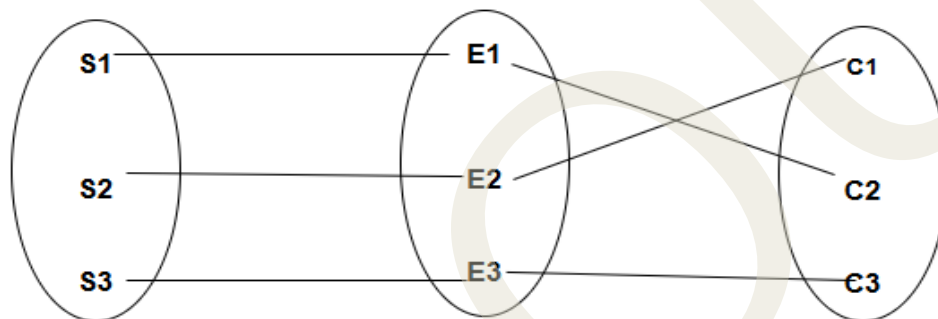


Fig. 1.3.14 Relationship set

Degree of a Relationship Set

The number of different entity sets participating in a relationship set is called the degree of a relationship set.

1. Unary Relationship: When there is only ONE entity set participating in a relation, the relationship is called a unary relationship. For example in fig 1.3.15, one person is married to only one person.

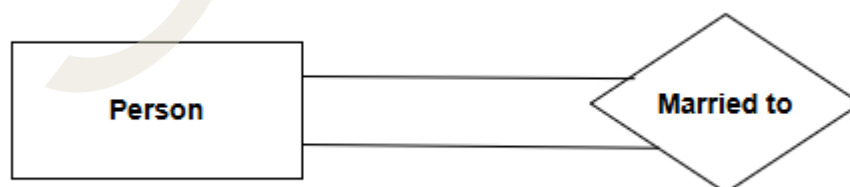


Fig. 1.3.15 Unary relationship

2. Binary Relationship: When there are TWO entities participating in a relationship, the relationship is called a binary relationship. For example in fig 1.3.16, a Student is enrolled in a Course.



Fig. 1.3.16 Binary relationship

3. Ternary Relationship: When there are three entity sets participating in a relationship, the relationship is called a ternary relationship. In a data model for an educational institution as shown in fig 1.3.17, a single relationship called "teaches" can link the entities Subject, Teacher, and Student. This relationship helps determine which teachers teach each subject and which students are enrolled in it.

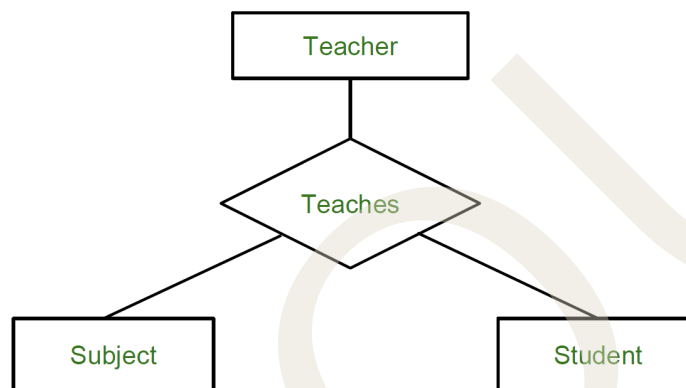


Fig. 1.3.17 Ternary relationship

4. N-ary Relationship: When there are n entities set participating in a relationship, the relationship is called an n-ary relationship.

Cardinality

The number of times an entity of an entity set participates in a relationship set is known as cardinality . Based on Cardinality relationships can be of different types:

- ◆ One to one
- ◆ One to many
- ◆ Many to one
- ◆ Many to many

a. One to one

A student has only one birth_certificate and a birth_certificate is given to one student. the student to bith_certificate relationship is one to one.



Fig. 1.3.18 One to One relationship

b. One to many

A student can borrow many books but a book can be borrowed by only one student. This student to book relationship is one to many.

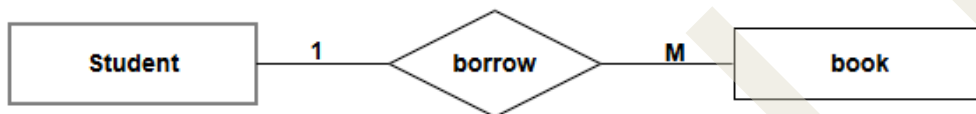


Fig. 1.3.19 One to many relationship

c. Many to one

Many students can enrol in a single department. But a single student cannot enroll in multiple departments at the same time

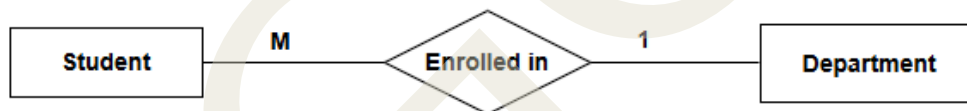


Fig. 1.3.20 Many to one relationship

d. Many to many

Many students can teach by many faculties and many faculties can teach many students.



Fig. 1.3.21 Many to many relationship

1.3.2.5 Generalization and Specialization

Generalization: Combining two or more entities into a higher-level entity. Example Car and trucks can be generalized into vehicles.

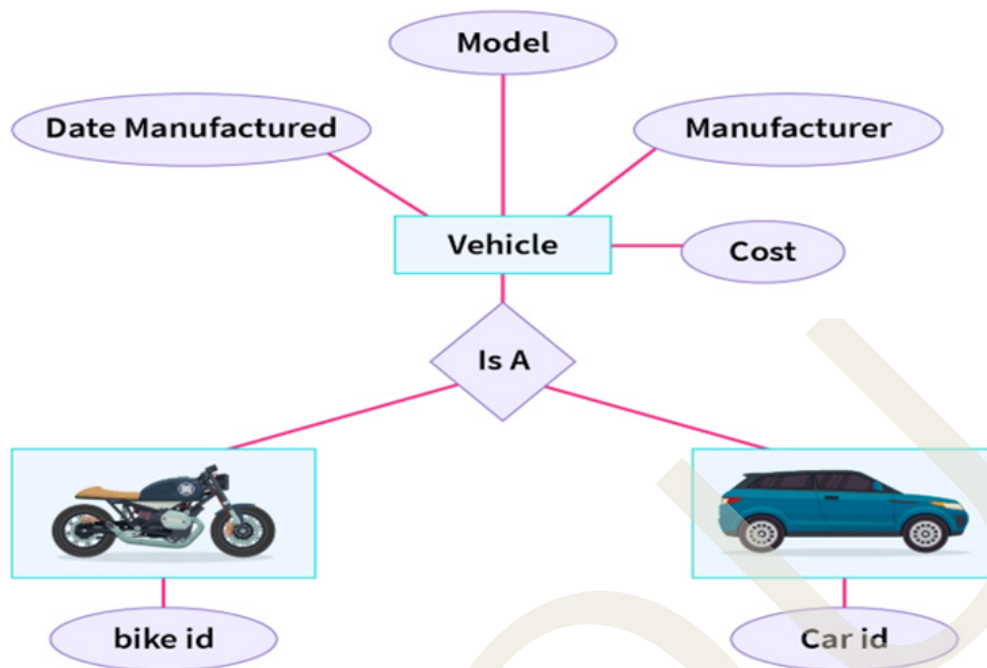


Fig. 1.3.22 Generalization of vehicle entity

Specialization: Creating sub-entities from a higher-level entity. Example Employee can be specialized into Faculty and admin_staff).

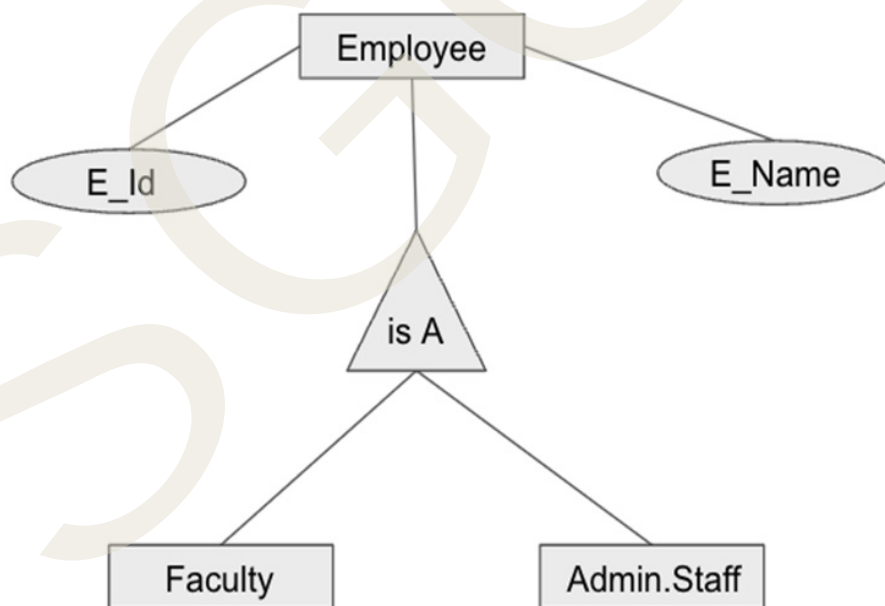


Fig. 1.3.23 Specialization of employee entity

1.3.2.6 Aggregation

Aggregation is an abstract relationship where a relationship itself is treated as an entity. It is useful when one relationship depends on another relationship. A Loan entity

is related to a Customer and a Bank, but the relationship between Bank and Loan can be further aggregated to represent a Loan Approval process.

1.3.2.7 Participation Constraint

Participation Constraint is applied to the entity participating in the relationship set.

1. Total Participation : Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of students will be total. Total participation is shown by a double line in the ER diagram.



Fig. 1.3.24 Total participation and partial participation

2. Partial Participation: The entity in the entity set may or may NOT participate in the relationship. If some courses are not enrolled by any of the students, the participation in the course will be partial.

The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.



Fig. 1.3.25 Partial participation

Using Set, it can be represented as,

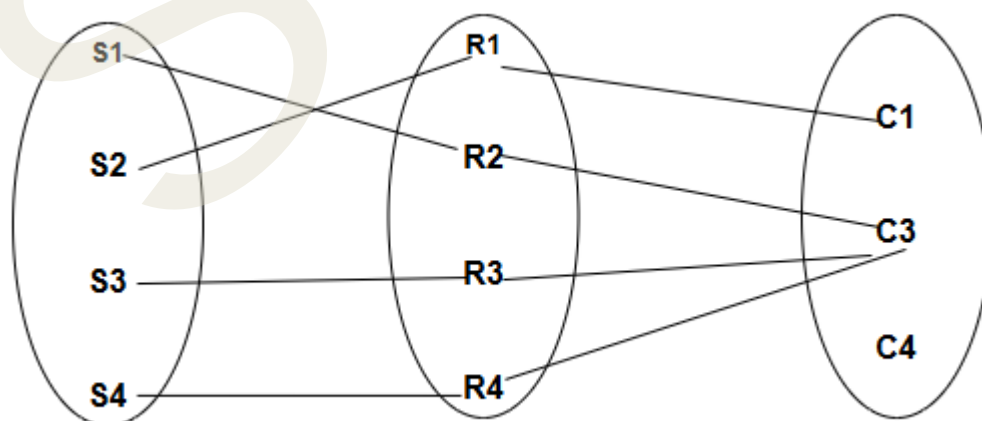


Fig. 1.3.26 Set representation of total participation and partial participation

Every student in the Student Entity set participates in a relationship but there exists a course C4 that is not taking part in the relationship.

1.3.3 Example of ER Model

Let us discuss the ER Model with the example of the University Academic Management System.

The system includes:

- ◆ Student information
- ◆ Faculty information
- ◆ University
- ◆ Classes
- ◆ Subject

In this student, faculty, classes and subjects are entities. A real world object is termed as an entity in a DBMS. An entity set is a group of similar entities and these entities can have attributes.

A student is an entity or a real world object with properties stud_id, stud_name and age. These properties are called attributes of a student entity. Similarly universities have attributes univ_id, univ_name etc.

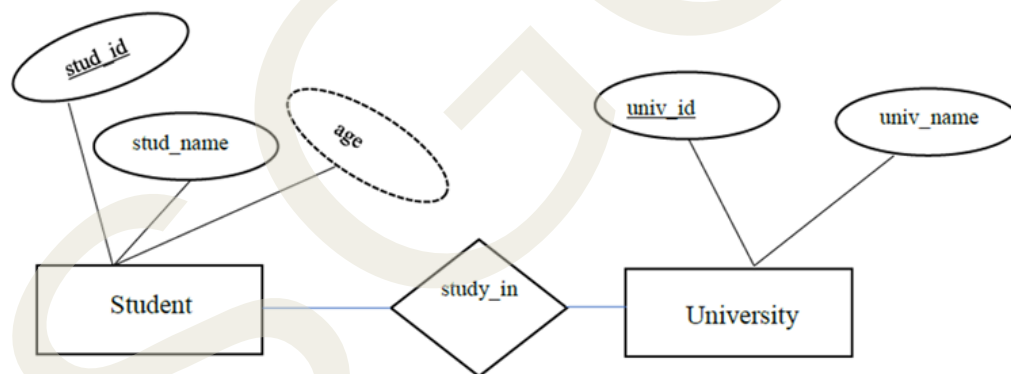


Fig. 1.3.27 Sample ER diagram with two entities: student and university.

Example : ER model for Hospital Management System

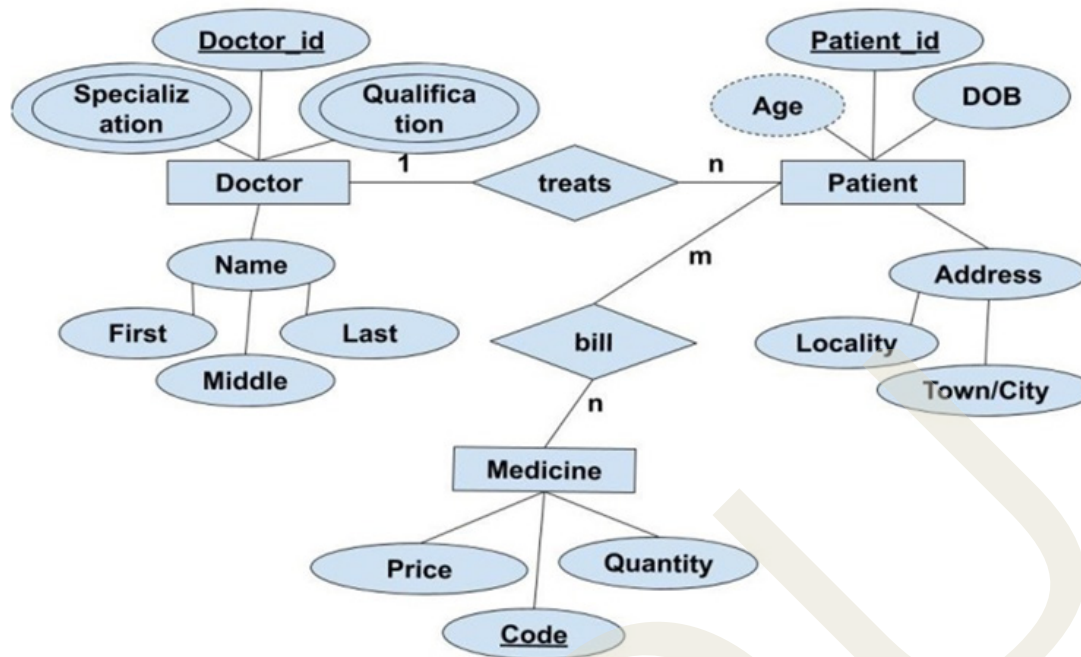


Fig. 1.3.28 ER model for hospital management system

Entities and Attributes

1. Doctor

- Doctor_ID (Primary Key)
- Name (First, Middle, Last)
- Specialization
- Qualification

2. Patient

- Patient_ID (Primary Key)
- Name
- Age
- DOB
- Address (Locality, Town/City)

3. Medicine

- Code (Primary Key)
- Price
- Quantity

4. Bill
 - Bill_ID (Primary Key)

Relationships

1. Treats (1:N relationship between Doctor and Patient)
 - A doctor can treat multiple patients, but each patient is treated by only one doctor.
2. Bill (M:N relationship between Patient and Medicine)
 - A patient can have multiple medicines in a bill, and a medicine can be billed for multiple patients.

1.3.4 Advantages of the ER Model

1. **Simple and Clear Representation:** ER diagrams provide an intuitive and visual way to represent data and relationships, making it easier to understand.
2. **Efficient Database Design:** Helps in structuring and organizing data efficiently before implementing the database.
3. **Flexibility:** Can be easily modified and extended to accommodate changes in requirements.
4. **Better Communication:** Acts as a bridge between database designers, developers, and stakeholders by providing a clear visual representation.
5. **Standardized Notation:** Uses well-defined symbols (entities, attributes, relationships) for clarity.
6. **Logical and Conceptual Design:** Helps in defining business rules and constraints before moving to physical database design.

1.3.5 Disadvantages of the ER Model

1. **Limited Representation of Complex Data** – ER models struggle with representing complex data types, inheritance, and advanced constraints.
2. **Not Suitable for Small Databases** – Can be overkill for simple databases with fewer relationships.
3. **Lack of Implementation Details** – Does not specify how data will be stored physically, requiring additional steps for database implementation.
4. **Challenging for Large Systems** – As complexity increases, ER diagrams can become too large and difficult to manage.
5. **Normalization Issues** – Sometimes, relationships may need further refinement using normalization techniques to avoid redundancy.
6. **Ambiguity in Many-to-Many Relationships** – Requires additional resolution (e.g., junction tables) to implement many-to-many relationships properly.

Recap

Components of ER Model

- ◆ Entities- Real-world objects (physical/conceptual)
- ◆ Entity Set - Collection of entities of the same type

Types of Entities

- ◆ Strong Entity – Has a primary key, can exist independently
- ◆ Weak Entity – Depends on a strong entity, lacks a primary key

Attributes in ER Model - Characteristics of an entity or relationship

- ◆ Types of Attributes:
- ◆ Simple Attribute – Atomic values
- ◆ Composite Attribute – Made of multiple simple attributes
- ◆ Derived Attribute – Computed from other attributes
- ◆ Single-Valued Attribute – Holds one value
- ◆ Multi-Valued Attribute – Holds multiple values

Keys in DBMS - Uniquely identify records and establish relationships

- ◆ Types of Keys
- ◆ Candidate Key : Minimal set of attributes uniquely identifying a tuple
- ◆ Super Key : Superset of a candidate key, supports NULL values
- ◆ Primary Key : Chosen candidate key, uniquely identifies records
- ◆ Composite Key : Combination of multiple attributes as a key
- ◆ Foreign Key : Attribute in one table referring to primary key in another

Degree of a Relationship Set

- ◆ Unary Relationship
- ◆ Binary Relationship
- ◆ Ternary Relationship
- ◆ N-ary Relationship

Cardinality in Relationships

- ◆ One-to-One Relationship
- ◆ One-to-Many Relationship
- ◆ Many-to-One Relationship
- ◆ Many-to-Many Relationship

Objective Type Questions

1. The relationship between entities in a data model can be represented using which Model?
2. What is the term used for real world objects in the ER model?
3. Which is the term used for an entity that can only exist when owned by another entity?
4. Which shape is used for denoting entities?
5. Which shape is used for denoting attributes?
6. Which of the following is a key attribute?
7. Which shape is used for denoting relationships?
8. Which shape is used for denoting links?
9. Which type of attribute can have multiple values?
10. Which shape is used for denoting multi-valued attributes?
11. Which shape is used for denoting derived attributes?
12. Generalization in an ER model represents:
13. Which shape is used for denoting total participation?
14. Which shape is used for denoting a weak entity set?
15. How can we represent a Primary key attribute?

Answers to Objective Type Questions

1. ER Model.
2. Entity
3. Weak entity
4. Rectangle
5. Ellipse
6. An attribute that uniquely identifies an entity
7. Diamond
8. Lines
9. Multi-valued attribute
10. Double ellipse
11. Dashed ellipse
12. The process of combining multiple entities into a higher-level entity
13. Double lines
14. Double rectangle
15. Under line

Assignments

1. Explain the components of an ER model with examples. Include entities, attributes, relationships, and constraints.
2. Describe different types of attributes in an ER model. Provide examples of simple, composite, derived, and multi-valued attributes.
3. Differentiate between strong and weak entities. Explain how weak entities are represented in an ER diagram with an example.
4. Discuss the types of relationships (one-to-one, one-to-many, many-to-many) in the ER model. Give a real-world example for each.
5. Explain the concepts of generalization, specialization, and aggregation in ER modeling with examples.

6. Draw an ER diagram for a University database that includes entities such as Students, Courses, Professors, and Departments. Indicate primary keys and relationships.
7. Convert the following scenario into an ER diagram: A hospital has doctors, patients, and nurses. Each patient is treated by one doctor, and a doctor can treat multiple patients. Nurses assist doctors, and each nurse is assigned to one doctor.
8. Identify the entities, attributes, and relationships for an online shopping system that includes Customers, Orders, Products, and Payments. Create an ER diagram for the system.
9. Discuss the advantages and limitations of the ER model. Suggest ways to overcome its limitations in real-world database design.
10. Convert an ER model into relational tables. Given an ER diagram for a Library Management System, define the relational schema with primary and foreign keys.

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2014). *Database management systems* (2nd ed.). McGraw-Hill Education.
2. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). *Database system concepts* (6th ed.). McGraw-Hill Education.

Reference

1. <https://www.geeksforgeeks.org/dbms>
2. https://www.tutorialspoint.com/dbms/dbms_architecture.htm

Unit 4

ER Model to Database Schema

Learning Outcomes

At the conclusion of this unit, the learner will be able to:

- ◆ identify ER diagrams with various types of entities
- ◆ convert an ER model with different relationship types into a relational schema
- ◆ understand the concept of total participation in an ER diagram
- ◆ explore methods for converting multivalued attributes into a relational schema

Prerequisites

In the previous lessons, you explored Entity-Relationship (ER) models, understanding how entities, attributes, and relationships form the foundation of database design. You learned how to represent real-world scenarios using ER diagrams, but have you ever wondered how these diagrams transform into actual databases? How does a well-structured ER model ensure a database is efficient, scalable, and free from redundancy? That's exactly what we'll uncover in this unit!

As you step into the world of ER model to database schema conversion, you'll discover how relationships like one-to-one (1-1), one-to-many (1-M), and many-to-many (M-N) shape relational tables. You will see how Primary Keys (PK) and Foreign Keys (FK) enforce data integrity, ensuring that records remain connected and meaningful. We'll also explore how to handle weak entities, identifying relationships, and multivalued attributes, making sure no piece of information is lost in translation.

By the end of this unit, you'll not only understand the theory behind database schema conversion but also gain the skills to design structured, optimized databases from ER models. Get ready to bridge the gap between conceptual design and practical implementation, because databases power everything from banking systems to social media, and mastering this process is your key to building robust, real-world applications!



Keywords

Foreign key, arity, cardinality, one to one, one to many, many to many, total participation.

Discussion

1.4.1 Relational Model

The Relational Model represents data and their relationships through a collection of tables. Each table also known as a relation consists of rows and columns. Every column has a unique name and corresponds to a specific attribute, while each row contains a set of related data values representing a real-world entity or relationship. This model is part of the record-based models which structure data in fixed-format records each belonging to a particular type with a defined set of attributes.

E.F. Codd introduced the Relational Model to organize data as relations or tables. After creating the conceptual design of a database using an ER diagram, this design must be transformed into a relational model which can then be implemented using relational database systems like Oracle SQL or MySQL.

The relational model represents DB in the form of a collection of various relations. This relation refers to a table of various values. And every row present in the table happens to denote some real-world entities or relationships. The names of tables and columns help us interpret the meaning of the values present in every row of the table. This data gets represented in the form of a set of various relations. Thus, in the relational model, basically, this data is stored in the form of tables. However, this data's physical storage is independent of its logical organisation.

Popular Relational Database Management Systems

- ◆ IBM – DB2 and Informix Dynamic Server
- ◆ Oracle – Oracle and RDB
- ◆ Microsoft – SQL Server and Access
- ◆ Properties of a Relational Model

The relational databases consist of the following properties

1. Every row is unique
2. All of the values present in a column hold the same data type
3. Values are atomic
4. The columns sequence is not significant
5. The rows sequence is not significant
6. The name of every column is unique



1.4.2 Key Concepts of the Relational Model

1. Relation (Table)

A **relation** refers to a **table** that organizes data into rows and columns. Each relation represents an entity or a real-world object, and has a name with **tuples (rows)** storing individual records and **attributes (columns)** defining the properties of those records.

Example : Course Relation

Table 1.4.1 Course Relation

Course_ID	Course_Name	Department	Duration (Years)
C101	Computer Science	Science & Technology	3
C102	Biology	Life Sciences	3
C103	Commerce	Business Studies	3
C104	Mathematics	Science & Technology	3
C105	Physics	Science & Technology	3

2. Field

A field is the smallest unit of data in a database. It represents a single data item within a table. In a "Student" table, a field can be "S101", "Surya S", "Commerce" are examples of fields.

3. Tuple (Row)

It is a single row of a table that consists of a single record. The relation above consists of five tuples, one of which is like:

C101	Computer Science	Science & Technology	3
------	------------------	----------------------	---

4. Attribute (Column)

It refers to every column present in a table. The attributes refer to the properties that help us define a relation. Each attribute has a unique name and a specified data type. Above relation contain four attributes, one of which is like:

CourseId
C101
C102
C103
C104
C105

5. Domain

A domain refers to the set of valid values that an attribute (column) can take. It defines the permissible data types and constraints for a given attribute.

Course_ID → Domain: Varchar (Positive Numbers Only)

Course_Name → Domain: Text (String with a max length, e.g., 50 characters)

Duration → Domain: Integer (Between 16 and 22 only)

6. Relation Schema

A relation schema defines the structure of the relation and represents the name of the relation with its attributes. e.g. STUDENT (Stud_Id, NAME, Age, Course) is the relation schema for STUDENT. The schema specifies the

- ◆ Relation name
- ◆ The name of each column or attribute.
- ◆ The domain of each field.

7. Relation Instance

The set of tuples of a relation at a particular instance of time is called a relation instance. It can change whenever there is an insertion, deletion or update in the database.

8. Degree

The number of attributes in the relation is known as the degree of the relation. The STUDENT relation defined above has degree 4.

9. Cardinality

The number of tuples in a relation is known as cardinality. The STUDENT relation defined above has cardinality 5.

10. NULL Values

The value which is not known or unavailable is called a NULL value. It is represented by NULL. e.g. PHONE of STUDENT having ROLL_NO 4 is NULL

Table 1.4.2

ROLL_NO	NAME	AGE	COURSE	PHONE
1	Anu S	16	Computer Science	9876543210
2	Daya M	17	Biology	8765432109
3	Rahul G	17	Commerce	7654321098
4	Surya S	16	Biology	NULL
5	Hari A	18	Computer Science	6543210987

11. Key

A key is an attribute or a set of attributes that uniquely identifies each tuple in a relation. In a Student table, "Student_ID" can be a primary key.

12. Constraints

Rules enforced to maintain the integrity of data in the database. Primary Key Constraint, Foreign Key Constraint, NOT NULL, UNIQUE, etc.

1.4.3 Constraints in Relational Model

While we design a Relational Model, we have to define some conditions that must hold for the data present in a database. These are known as constraints. One has to check these constraints before performing any operation (like insertion, updating and deletion) in the database. If there occurs any kind of a violation in any of the constraints, the operation will ultimately fail.

1.4.3.1 Domain Constraints

The domain constraints are like attribute level constraints. Now an attribute is only capable of taking values that lie inside the domain range. For example, if a constraint $ID_NO > 0$ is applied on the EMPLOYEE relation, inserting some negative value of ID_NO will result in failure.

1.4.3.2 Key Integrity

Each and every relation present in the database should have at least one set of attributes that uniquely defines a tuple. Those sets of attributes are known as keys. For example, ID_NO in EMPLOYEE is a key. Now, remember that no two students would be capable of having the very same ID number. Thus, a key primarily consists of these two properties:

- ◆ It has to be unique for all the available tuples.
- ◆ It can not consist of any NULL values.
- ◆ Referential Integrity

Whenever one of the attributes of a relation is capable of only taking values from another attribute of the same relation or other relations, it is termed referential integrity.

1.4.4 Relation Schema

A Relational Schema is like a blueprint for a relational database. It defines how data is organized into tables (relations) and how these tables are connected through keys and constraints.

Think of it like an architectural plan for a building—before constructing a house, an architect designs a layout showing rooms, doors, and connections. Similarly, a relational schema defines the structure of the database before storing actual data. e.g. STUDENT (Stud_Id, NAME, Age, Course) is the relation schema for STUDENT. The

schema specifies the

13. Relation name
14. The name of each column or attribute.
15. The domain of each field.

1. Relational Name

Name of the table that is stored in the database. It should be unique and related to the data that is stored in the table. For example- The name of the table can be Employee store the data of the employee.

2. The name of each column or attribute

Attributes specify the name of each column within the table. Each attribute has a specific data type.

3. The domain of each field

The set of possible values for each attribute. It specifies the type of data that can be stored in each column or attribute, such as integer, string, or date.

4. Primary Key

The primary key is the key that uniquely identifies each tuple. It should be unique and not be null.

5. Foreign Key

The foreign key is the key that is used to connect two tables. It refers to the primary key of another table.

6. Constraints

Rules that ensure the integrity and validity of the data. Common constraints include NOT NULL, UNIQUE, CHECK, and DEFAULT.

Example: Student (stud_id: string, stud_name: string, login: string, age: integer, gpa : real)

Table 1.4.3 Student relation

Stud_id	Stud_name	login	age	gpa
2100	Smith	Smith@cs	20	4.2
2155	Jack	Jack@hindi	21	3.9
2345	Nimal	Nimal@english	20	3.5
2454	Job	Job@history	22	3.8

In the above schema, stud_id, stud_name, login have a domain named string, age has integer and gpa has real.

A relation instance in a database management system (DBMS) is a set of tuples in a relation at a specific moment in time. A relation instance can change if data is added, deleted, or updated in the database.

When accessing records in a Student relation, `stud_id = 2100`, `stud_name = smith`, `login = Smith@cs`, `age = 20`, `gpa = 4.2` which will return a single tuple or row. That is “Smitha, Smith@cs, 20, 4.2” is an example for a single tuple or row in a relational database.

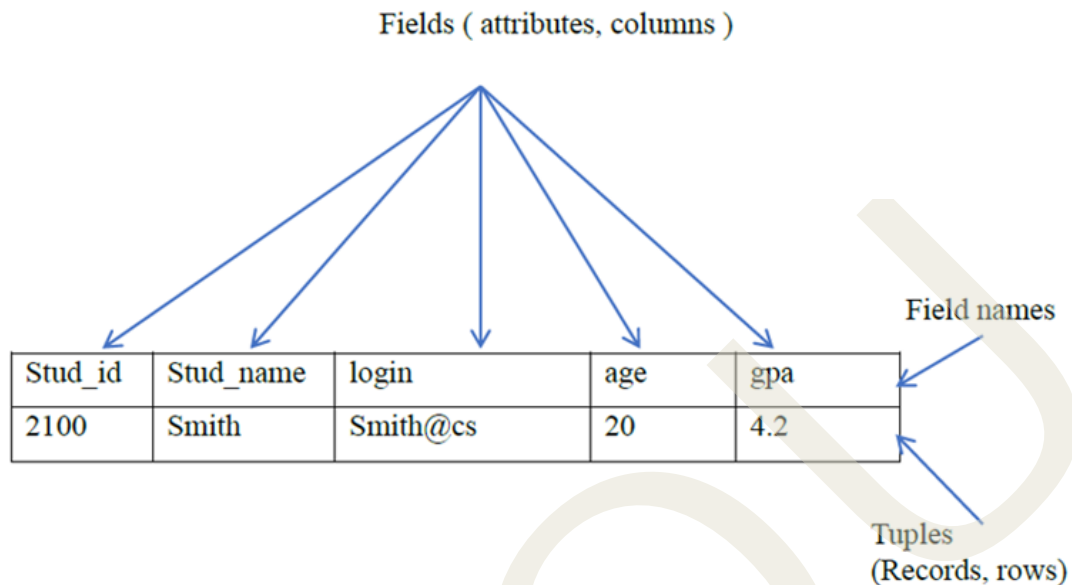


Fig. 1.4.1 Fields and attributes

Degree of a relation: The number of attributes in a relation represents the degree or arity of the relation. Table “Student” has degree five.

Cardinality : The number of tuples or rows represents the cardinality of a relation instance.

Table “Student” has cardinality four.

A relational Database has a collection of relations with distinct relation names. For example university databases have relations named student, faculty, courses, classes etc.

Foreign key constraints

If information stored in a relation is linked to another relation, the integrity constraint used is foreign key.

The Enrolled table

Enrolled (`stud_id`: string, `course_id`: string, `grade`: string)

For ensuring that only registered students are enrolled in courses. The values in the `stud_id` field of enrolled should match the `stud_id` field of student relation. The `stud_id` field of enrolled is called a foreign key and refers to student relation. The foreign key in the Enrolled relation (`stud_id` in enrolled relation) must match the primary key of the referenced relation (student relation). Name of the attribute can differ and datatypes need to be compatible.

1.4.5 Conversion of strong entity into a relational relation

In the Entity-Relationship (ER) model, a strong entity is an entity that can exist independently in the database and has a primary key that uniquely identifies each of its instances. When converting a strong entity into a relational database table, follow these steps:

Steps for Conversion:

- ◆ Create a Table:
 - Each strong entity is converted into a separate relation (table).
- ◆ Include Attributes as Columns:
 - All attributes of the entity become columns in the table.
 - Multivalued attributes can be ignored
 - Composite attribute in the ER diagram should be divided to simple attributes and included in the table
- ◆ Select a Primary Key:
 - The primary key of the strong entity becomes the primary key of the corresponding table.

Consider this ER diagram shown in figure 1.4.2

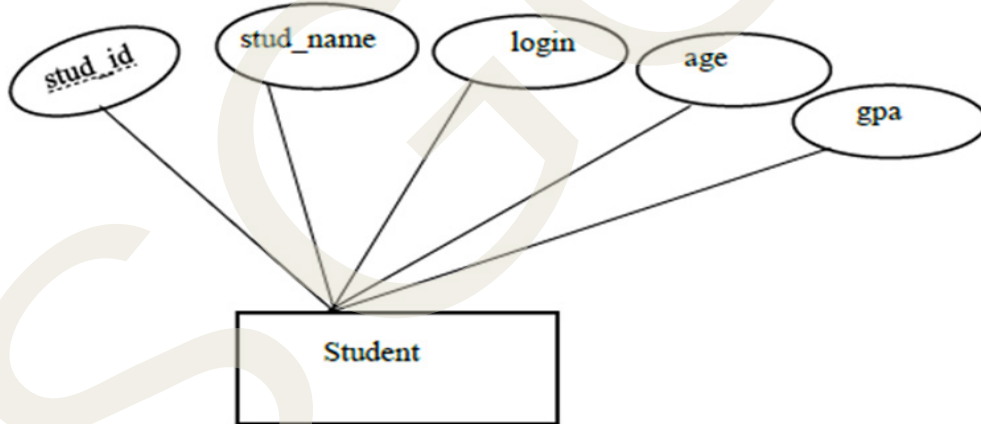


Fig. 1.4.2 ER diagram of student

In the below example faculty is a strong entity with attributes fac_id as primary key, fac_name, email_id.

In this example a student is a strong entity with attributes stud_id as primary key, stud_name, login, age, gpa.

1.4.5.1 Conversion of ER Model to Relational Schema

Table 1.4.4 Student relation 2

Stud_id	Stud_fname	Stud_lname	login	age	gpa
2100	Neo	Smith	Smith@cs	20	4.2
2155	Harry	Jack	Jack@ economics	21	3.9
2345	John	Nimal	Nimal@ english	20	3.5
2454	Dennis	Job	Job@ history	22	3.8

Schema of the above relation is:

Student (stud_id: string, stud_fname: string, stud_lname: string, login: string, age: integer, gpa : real)

In the student table all the strong entity conversion rules are followed ie. ,

- ◆ No change in strong entity name
- ◆ All entities in ER diagram included in student table
- ◆ Composite attribute name is divided into simple attributes Stud_fname and stud_lname.
- ◆ No multi valued attributes present in the table.
- ◆ Stud_id attribute selected as the Primary Key.

1.4.6 Conversion of weak entity into a relational relation

A **weak entity** is an entity that **cannot exist independently** and relies on a **strong entity** for its existence. It does not have a **sufficient primary key** on its own and instead uses a **foreign key** referencing the related strong entity.

Steps for Conversion:

- ◆ Create a Separate Table for the Weak Entity
 - Convert the weak entity into a relation (table).
- ◆ Include Attributes as Columns
 - All attributes of the weak entity become columns in the table.
- ◆ Use a Composite Primary Key
 - A weak entity does not have a unique identifier, so its primary key is a combination of:

- Primary key of the strong entity must be included as the foreign key of the weak entity
 - The discriminator (partial key) of the weak entity.
- ◆ Establish a Foreign Key Relationship
- Choose the primary key as a combination of foreign key and discriminator attribute of the weak entity

Notes

Weak entities need a strong entity to exist. A composite primary key is formed using the foreign key + a unique attribute of the weak entity. A foreign key constraint ensures referential integrity.

Example

Consider this ER diagram with loan entity and payment entity

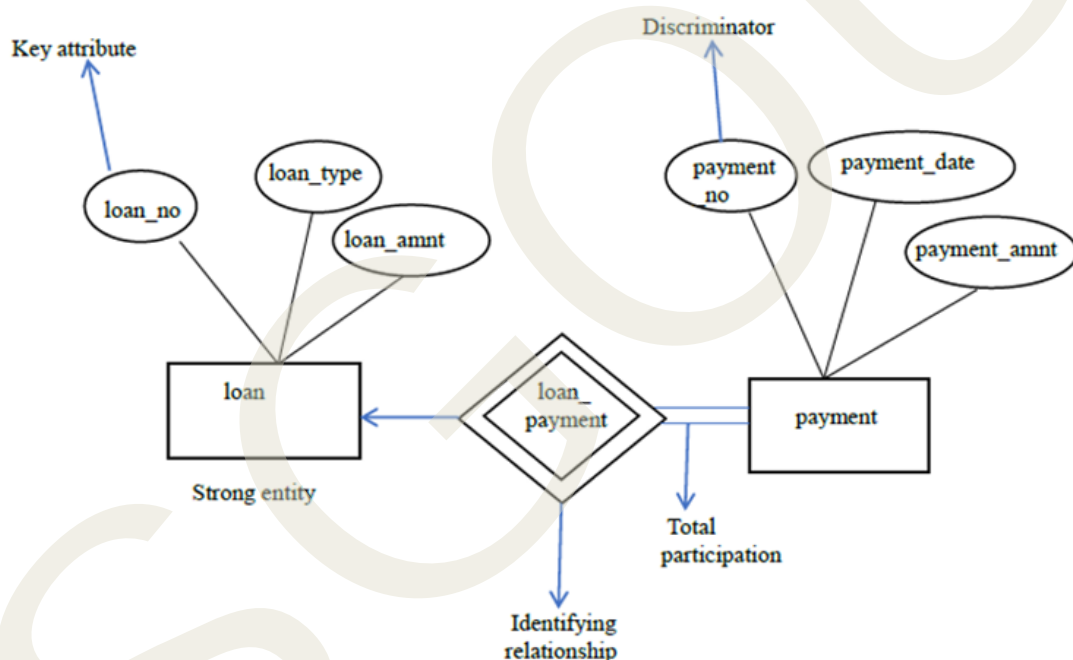


Fig. 1.4.3 ER diagram of loan

Loan is a strong entity and payment is a weak entity. The Discriminator or the partial key for a payment entity is payment_no attribute. The partial key along with the key attribute of the loan helps in identifying each payment record uniquely. Strong and weak entity sets always have parent-child relationships.

1.4.7 Convert Relationships to Tables

1.4.7.1 Conversion of a 1:1 Relationship into a Relational Table

A 1:1 (one-to-one) relationship in an ER model means that for each instance of Entity A, there is at most one corresponding instance of Entity B, and vice versa.

Steps for Conversion:

1. Merge into a Single Table (if Possible)
 - If both entities have the same primary key, they can be combined into a single table.
 - This avoids redundancy.
2. Use Separate Tables with a Foreign Key
 - If merging is not ideal, one entity's primary key is added as a foreign key in the other.
 - The foreign key should have a unique constraint to maintain the 1:1 relationship.
3. Foreign Key Placement
 - Choose the entity with fewer instances to hold the foreign key.

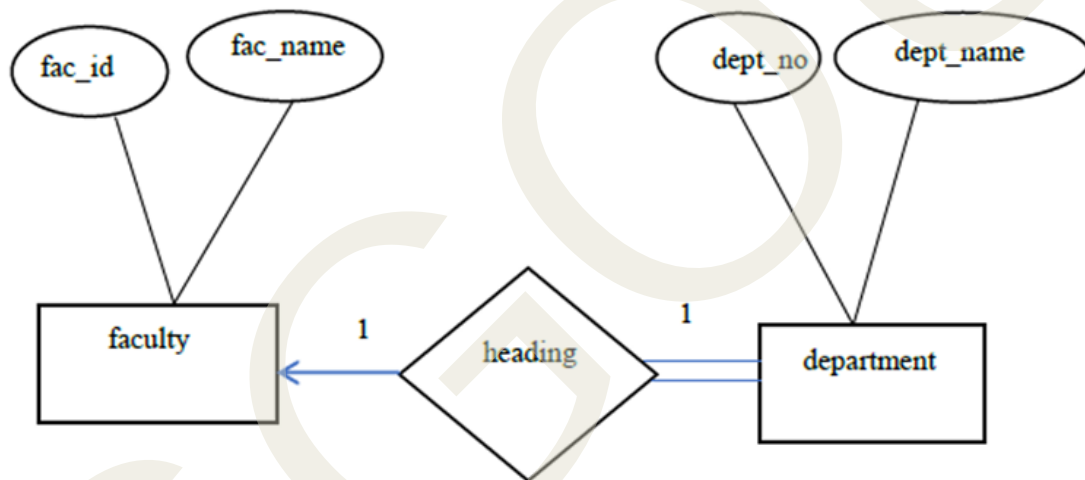


Fig. 1.4.4 1:1 Relationship

In this example if we are making dept_no as foreign key of the faculty table, then we can't say that all faculties are heading departments. So it is convenient to make fac_id as foreign key to the department table, i.e., for every department there exists a head of the department, which represents the total participation.

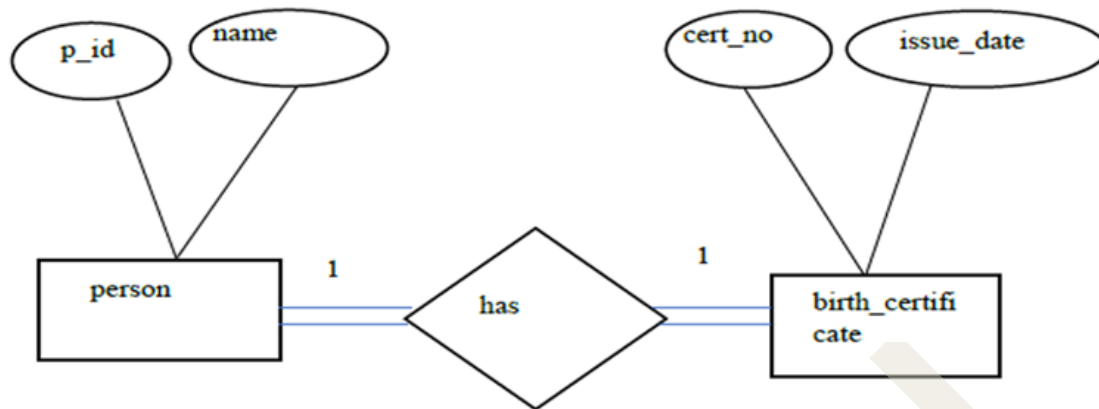


Fig. 1.4.5 1:1 Relationship(Total participation)

In this one to one relationship both entities have total participation so that both tables can be merged to a single table.

1.4.7.2 Conversion of one-many(1-M) relationship

In a one-to-many (1-M) relationship in a database, one record in the first table can be related to multiple records in the second table. The conversion of a 1-M relationship typically involves the following steps:

- a. Identifying Entities and Relationship
 - A 1-M relationship occurs when one entity (say A) is related to multiple occurrences of another entity (say B).
- b. Conversion to Tables in a Relational Database
 - The "one" side becomes a table with a Primary Key (PK).
 - The "many" side becomes another table, containing:
 - Its own Primary Key (PK)
 - A Foreign Key (FK) referencing the PK of the "one" side.
- c. Referential Integrity
 - The Foreign Key (FK) in the Faculty table ensures that each dep_id is linked to a valid fac_id.

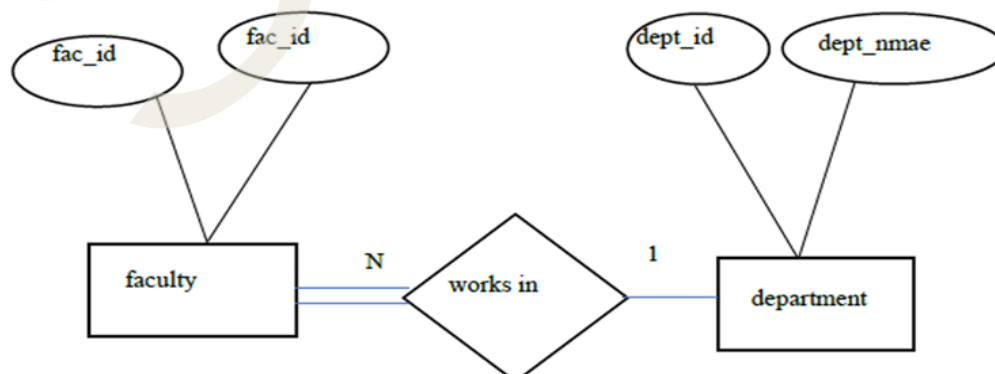


Fig. 1.4.6 1-M relationship

In this one too many examples, dept_id can be made as foreign key for the faculty table. If we are making fac_id as foreign key to the department table, the fac_id will be multi-valued so it needs to be avoided.

1.4.7.3 Conversion of Many-to-Many (M-N) Relationship

A many-to-many (M-N) relationship exists when multiple records in one entity are related to multiple records in another entity. This type of relationship cannot be directly implemented in a relational database. Instead, it is converted into two one-to-many (1-M) relationships using a junction (bridge) table.

Identifying Entities and Relationship

- ◆ Suppose we have two entities:
 - Students (who can enroll in multiple courses)
 - Courses (which can have multiple students enrolled)
- ◆ Since a student can enroll in multiple courses, and a course can have multiple students, this forms an M-N relationship.

Conversion Strategy: Create a Junction Table

- ◆ Create two one-to-many (1-M) relationships using a new table (junction table).
- ◆ This new table contains Foreign Keys (FKs) referencing the primary keys of both entities.
- ◆ Primary Key: EnrollmentID uniquely identifies each enrollment.
- ◆ Foreign Keys: StudentID and CourseID link to the respective tables.
- ◆ ON DELETE CASCADE: Ensures that if a student or course is deleted, associated enrollments are also removed.

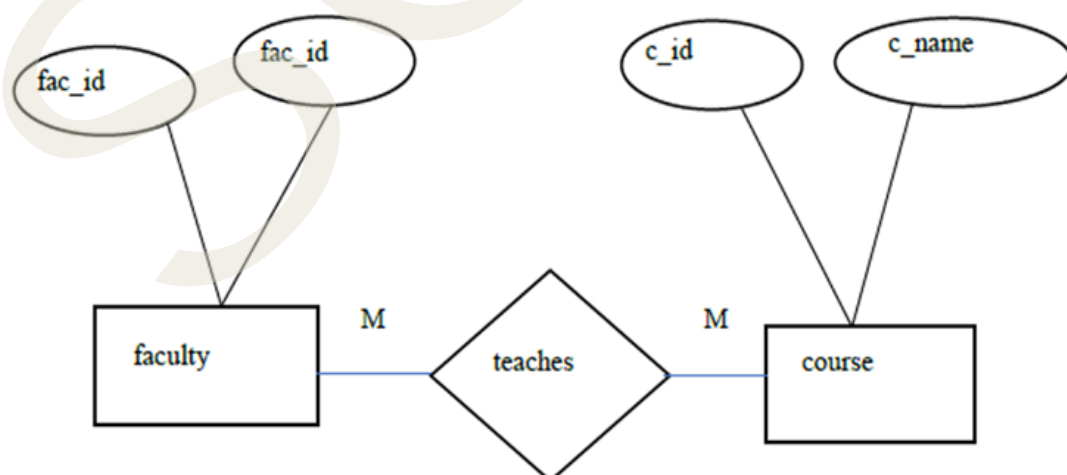


Fig. 1.4.7 M:M relationship

In this example if we need to add hours_handled then we need to create a new table

with selected attributes from both tables, such as fac_id, c_id, hours_handled .

Conversion of multivalued attributes

Create a separate table for each multivalued attribute and include the primary key of the current table as foreign key of new table

The combination of foreign keys and multivalued attributes must be considered as the primary key of the new table

Recap

- ◆ Concepts of Relational Database Model
- ◆ Components of Relational Database model
 - Relation
 - Attribute
 - Tuple
 - Domain
 - Relation Instance
 - Degree
 - Cardinality
 - Key
 - Constraints
- ◆ Concepts of foreign key reference.
- ◆ Strong entity to a table: separate table for strong entity-ignore multivalued attributes - composite attribute to simple attributes
- ◆ Weak entity to table: separate table for strong entity - primary key of strong entity as foreign key
- ◆ 1-1 relationship to table: primary key of one of the sides as foreign key
- ◆ 1-M relationship to table: the table of M side should have the primary key of the other side as a foreign key
- ◆ M-N relationship to table: create a separate table - include the primary key of M side and N sides
- ◆ Multivalued attributes: separate table for each multivalued attribute

Objective Type Questions

1. What is the term used to represent the number of fields in a relation?
2. Which is the name of the integrity constraint used to refer to another table?
3. What is the other name for the partial key?
4. Which is the type of participation that some of the entities of the corresponding entity type participate in the relationship?
5. Which is the term used for representing data in relational models?
6. Which is the type of participation that every entity of the corresponding entity type participates in the relationship?
7. Double line is used to represent which type of participation in a relationship?
8. What is the term used to represent the number of rows in a relation?
9. Which is the other name for a record in a relation?
10. Which is the name of the key contained in a weak entity?
11. Double diamond is used to represent which relationship?

Answers to Objective Type Questions

1. Degree
2. Foreign Key
3. Discriminator
4. Partial Participation
5. Relation
6. Total Participation
7. Total Participation
8. Cardinality
9. Tuple
10. Partial Key
11. Identifying Relationship

Assignments

1. Explain the key components of the Relational Database Model. How do they contribute to data organization and integrity?
2. Define and differentiate between Relation, Attribute, Tuple, and Domain in the context of a relational database. Provide examples.
3. What is a Relation Instance? How does it differ from Relation Schema? Illustrate with an example.
4. Explain the concepts of Degree and Cardinality in a relational database. How do they impact database design?
5. Discuss the different types of Keys (Primary Key, Candidate Key, Super Key) in a relational model. Why is a Primary Key important?
6. What are Constraints in a relational database? Explain different types of constraints with suitable examples.
7. What is a Foreign Key? How does foreign key reference help maintain referential integrity? Provide an example using two related tables.
8. Describe the process of converting a Strong Entity into a relational table. How are composite and multivalued attributes handled?
9. Explain how Weak Entities are converted into relational tables. Why is the Primary Key of the Strong Entity used as a Foreign Key?
10. Discuss the relational mapping rules for different types of relationships (1-1, 1-M, M-N). How are multivalued attributes handled in relational tables?

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2014). *Database management systems* (2nd ed.). McGraw-Hill Education.
2. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). *Database system concepts* (6th ed.). McGraw-Hill Education

Reference

1. <https://www.geeksforgeeks.org/dbms>
2. https://www.tutorialspoint.com/dbms/dbms_architecture.htm





Structured Query Language

Unit 1

Functional Dependency

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ define the concept of functional dependency
- ◆ explain the importance of functional dependency in databases
- ◆ introduce various types of functional dependencies
- ◆ create awareness about the necessity of normalization
- ◆ familiarize students with different normal forms

Prerequisites

When designing a database, one of the key challenges is determining the best way to group attributes within relations. The effectiveness of a particular grouping depends on how well it maintains data consistency, minimizes redundancy, and avoids anomalies. But how do we evaluate whether one grouping is better than another? How do we assess the quality of a relational schema design?

A major concern in relational databases is information redundancy, which leads to unnecessary storage consumption. Another issue is update anomalies, which occur in three forms: insertion, deletion, and modification anomalies. These anomalies arise when changes made to one part of the database are not properly reflected in related parts, leading to inconsistencies.

To ensure an efficient and well-structured database, it is important to design relations that minimize redundancy and prevent update anomalies. Functional dependency and normalization provide systematic approaches to achieving this goal, helping to structure data in a way that enhances integrity and efficiency.

Keywords

Functional dependency, full functional dependency, partial functional dependency, transitive functional dependency, normal forms



Discussion

2.1.1 Functional dependency : Concept and Significance

Consider the relation STUDENT(student id, student name, date of birth). Consider the attributes date of birth and student name. There might be multiple students with the same date of birth on the table. Therefore, we cannot uniquely identify a student's name from his/her date of birth. So we can say that the attribute date of birth does not determine the attribute student name or we can say student name is not functionally dependent on date of birth.

Now consider the attributes student id and student name. Every student will have a unique student id. Therefore, we can uniquely identify a student name from his/her student id. Then we can say that student id determines student name or student name is functionally dependent on student id.

Functional dependency: A functional dependency, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of relation R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t1 and t2 in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$. This means that the values of the X component of a tuple uniquely (or functionally) determine the values of the Y component.

Consider the relation R(A,B,C) given in Table 2.1.1.

Table 2.1.1 A relation R(Order ID, Customer Name, Item)

Order ID	Customer Name	Product
101	John	Laptop
101	John	Mouse
102	John	Laptop
102	John	Keyboard

Consider the following functional dependencies for the above relation R:

Order ID \rightarrow Customer Name

Order ID, Product \rightarrow Customer Name

Consider attributes Order ID and Customer Name. 101 and 102 are the distinct values of attribute Order ID in the table. For each VALUE OF 101, there is a unique value in B that is value as John. Therefore, Order ID \rightarrow Customer Name. i.e. Order ID determines Customer Name or we can say Customer Name is functionally dependent on Order ID.

Consider the values of the attributes Customer Name and Product. For each value 'john' in Customer Name there are multiple corresponding values in the Product column. Therefore, the functional dependency Customer Name \rightarrow Product cannot hold true.

Consider the attributes Order ID, Product and Customer Name. The values of Order ID, Product are (101, Laptop), (101, Mouse), (102, Laptop) and (102, Keyboard). For each combination of Order ID and Item, there is a unique value in Customer Name. Therefore, Order ID and Item functionally determines Customer Name. i.e. Order ID, Product \rightarrow Customer Name.

2.1.2 Types of functional dependencies

There are three types of functional dependencies.

1. Partial functional dependency
2. Transitive functional dependency
3. Full functional dependency

2.1.2.1 Partial functional dependency

Definition:

A partial functional dependency occurs when a non-prime attribute (an attribute that is not part of any candidate key) is functionally dependent on part of a composite candidate key, but not on the entire key.

Consider the relation $R(A, B, C, D)$ where AB is the primary key. Following functional dependencies are defined on the relation:

$$AB \rightarrow C$$

$$B \rightarrow D$$

D is a non key attribute as it is not part of the primary key AB . B is a part of the primary key AB . D is dependent on B . Therefore, $B \rightarrow D$ is a partial functional dependency as a non key attribute is dependent on part of the primary key.

All non-key attributes in a table should be totally dependent on the entire primary key. If they depend on only a part of the primary key, it is called a partial functional dependency.

Under the following circumstances a table cannot have partial functional dependencies.

- ♦ if primary key consists of only one attribute
- ♦ if table consists of only two attributes
- ♦ if all the attributes in a table are part of primary key

2.1.2.2 Transitive functional dependency

Consider the relation $R(A, B, C, D)$ where AB is the primary key. Following functional dependencies are defined on the relation:

$$AB \rightarrow C \text{ (Primary key } AB \text{ determines } C)$$

$C \rightarrow D$ (C determines D , where C and D are non key attributes)

Thus $AB \rightarrow D$ exists transitively. ie $AB \rightarrow D$ via C

C and D are non key attributes and C determines D . We have a transitive dependency. ie, D is transitively dependent on AB because it is not directly determined by AB , but rather through C .

A transitive functional dependency is a type of functional dependency where a non- key attribute depends on another non key attributes which in turn depends on the primary key.

2.1.2.3 Full functional dependency

A full functional dependency occurs in a relational database when a non-prime attribute is functionally dependent on the entire candidate key and not on any proper subset of it. This means the dependency holds only when all attributes of the candidate key are considered together, and removing any attribute from the candidate key would break the dependency.

Consider the functional dependency $ABC \rightarrow D$ where D is dependent on A, B and C . If any attribute is removed from ABC the resulting functional dependency becomes invalid. For example, If A is removed, the resulting dependency would be $BC \rightarrow D$. However, since D depends on all three attributes A, B and C , the dependency $BC \rightarrow D$ is invalid. Therefore, $ABC \rightarrow D$ represents a full functional dependency.

In general, a dependency of the form $X \rightarrow Y$, is considered a full functional dependency if the removal of any attribute from X makes the dependency $X \rightarrow Y$ invalid.

2.1.3 Database Anomalies

In a database, improper organization of data can lead to problems such as redundancy and anomalies. Let's understand these issues using the example of a STUDENT table.

Example Table: STUDENT

Attributes:

1. student_id: Unique ID for each student
2. student_name: Name of the student
3. department: The department in which the student is enrolled
4. dept_head: The head of the department

2.1.3.1 Redundancy

Definition:

When the same data is repeated unnecessarily in a database, it leads to redundancy.

Example:

Suppose there are 100 students in the BCA department. For all these students, the

values for the attributes department and dept_head will be repeated 100 times. This repetition is an example of redundancy, which increases storage requirements and makes the database inefficient.

2.1.3.2 Anomalies

Anomalies occur when there are issues in inserting, updating, or deleting data due to poor database design. The major types of anomalies are:

1. **Update Anomaly:** It happens when updating a single piece of information, requires multiple changes that leads to potential inconsistencies.

Example:

If the head of the BCA department changes, the change must be made for all 100 students in the STUDENT table. If we forget to update the dept_head for any of these students, the database will become inconsistent, as different records may show different department heads.

2. **Insertion Anomaly:** It is a database inconsistency that occurs when design of a table presents the insertion of certain data.

Example:

Suppose a new department is created, but there are no students yet in that department. To record the new department in the STUDENT table, we would have to insert NULL values for student_id and student_name. However, primary keys like student_id cannot be NULL. This situation is an example of an insertion anomaly.

3. **Deletion Anomaly:** A deletion anomaly occurs when deleting a record unintentionally removes important information that should have been retained.

Example:

If a department has only one student and that student discontinues, deleting the student's record would also remove the information about the department. This results in a loss of important data about the department.

2.1.3.3 Solving Anomalies with Normalization

To solve updation, insertion, deletion anomalies and minimize redundancies, we need a formal way of analysing the given relation. Normalization provides a formal framework for analysing relation schemas based on functional dependencies among attributes of a relation. Normalization checks whether a relation schema satisfies a normal form. A relation schema that does not satisfy the given normal form can be decomposed into smaller relation schemas that satisfies the normal form. In our case, we are considering the following normal forms.

- ◆ First normal form (1NF)
- ◆ Second normal form (2NF)

- ◆ Third normal form (3NF)
- ◆ Boyce-Codd normal form (BCNF)

2.1.4 First Normal Form

Definition:

A table is said to be in First Normal Form (1NF) if:

1. All the values in the table are atomic (indivisible).
2. Each column contains only one value for each row (no repeated or grouped values in a single cell).
3. Each row is unique, meaning it can be identified by a unique primary key.

Consider a relation STUDENT(Student_ID, Name, Subjects) as shown in the table 2.1.2

Table 2.1.2 STUDENT relation

Student_ID	Name	Subjects
1	John	Math, Physics
2	Alex	Chemistry, Biology
3	Helen	Math, Chemistry, Biology

In this table:

The Subjects column contains multiple values in every row. (such as "Math, Physics" for John), which is not atomic.

This violates 1NF, as each column must contain indivisible values.

Solution (Convert to 1NF):

To convert the relation into 1NF, we separate the multiple values in the Subjects column into individual rows, making sure that each cell holds only one value.

Table 2.1.3 Relation after converted to 1NF

Student_ID	Name	Subjects
1	John	Math
1	John	Physics
2	Alex	Chemistry
2	Alex	Biology
3	Helen	Math
3	Helen	Chemistry
3	Helen	Biology

Now, every column contains atomic values, and the relation is in 1NF.

In simple terms, First Normal Form (1NF) does not allow a relation to have other relations or attribute values within a tuple. It is used to remove multivalued attributes in a table, ensuring each attribute holds only a single value.

2.1.5 Second Normal Form

Second Normal Form (2NF) is a level of database normalization that ensures a table is organized to reduce redundancy and dependency. It builds upon the rules of First Normal Form (1NF). A table is in 2NF if it meets the following conditions:

1. It is already in First Normal Form (1NF)
2. It has no partial dependencies

Example 1:

Suppose we have the following relation as shown in table 2.1.4:

Table 2.1.4 Customer data

Customer_ ID	Order_ ID	Product_ ID	Product_ Name	Product_ Price
1	101	P1	Laptop	40000
1	102	P2	Phone	20000
2	103	P1	Laptop	40000
3	104	P3	Tablet	10000

Consider the following functional dependencies:

Customer_ID, Order_ID \rightarrow Product_ID, Product_Name, Product_Price

Product_ID \rightarrow Product_Name, Product_Price

The primary key of the relation is Customer_ID, Order_ID.

Product_Name and Product_Price depend only on Product_ID, not on the full primary key

Customer_ID, Order_ID. This is a partial dependency, which violates 2NF. So the above given relation is not in 2NF.

Solution: Convert to 2NF:

To convert the relation into 2NF, we need to remove the partial dependency by splitting the relation:

Relation 1: Store customer-order-product data that depends on the full primary key Customer_ID, Order_ID.

◆ R1(Customer_ID, Order_ID)

- ◆ Functional Dependency:

Customer_ID, Order_ID → Product_ID

Table 2.1.5 Relation R1

Customer_ID	Order_ID	Product_ID
1	101	P1
1	102	P2
2	103	P1
3	104	P3

Relation 2: Store product details that depend only on Product_ID.

- ◆ R2(Product_ID, Product_Name, Product_Price)

- ◆ Functional dependency:

Product_ID → Product_Name, Product_Price

Table 2.1.6 Relation R2

Product_ID	Product_Name	Product_Price
P1	Laptop	40000
P2	Phone	20000
P3	Tablet	10000

Now the relations, R1 and R2 are in 2NF.

2.1.6 Third Normal Form

Third Normal Form (3NF) is a way of organizing a database to remove unnecessary data duplication and ensure the data is logically stored.

A table is in 3NF if:

1. It is already in Second Normal Form (2NF).
2. It has no transitive dependencies.

Example:

Consider the following relation **Employee**

Table 2.1.7 Employee Data

Emp_ID	Emp_Name	Dept_ID	Dept_Name
1	Alice	D1	HR
2	Helen	D2	IT
3	John	D1	HR

- ◆ Emp_ID is the primary key.
- ◆ Emp_Name, Dept_ID, and Dept_Name are non-prime attributes.
- ◆ Functional Dependencies:

1. Emp_ID \rightarrow Emp_Name, Dept_ID

2. Dept_ID \rightarrow Dept_Name

The given relation Employee is not in 3NF. Because the non-prime attribute Dept_Name depends on another non-prime attribute Dept_ID, which creates a transitive dependency:

ie, nonprime attribute Dept_Name indirectly depends on Emp_ID

- ◆ Emp_ID \rightarrow Dept_ID \rightarrow Dept_Name

According to 3NF, all non-prime attributes should depend only on the primary key and not on other non-prime attributes.

Solution to Convert into 3NF:

Break the table into two smaller tables:

1. Employee Table

Table 2.1.8 Employee table

Emp_ID	Emp_Name	Dept_ID
1	Alice	D1
2	Helen	D2
3	John	D1

2. Department Table

Table 2.1.9 Department Table

Dept_ID	Dept_Name
D1	HR
D2	IT

- ◆ In the Employee Table, all non-prime attributes (Emp_Name, Dept_ID) depend directly on the primary key (Emp_ID).
- ◆ In the Department Table, the non-prime attribute (Dept_Name) depends directly on the primary key (Dept_ID).
- ◆ There are no transitive dependencies. So the relations are in 3NF.

2.1.7 Boyce-Codd Normal Form

BCNF is a higher version of the Third Normal Form (3NF). It resolves anomalies caused by certain types of functional dependencies. A relation is in BCNF if, for every

functional dependency $X \rightarrow Y$, X is a superkey.

Key Characteristics:

- ◆ A superkey is any attribute or combination of attributes that can uniquely identify a row in a table.
- ◆ BCNF eliminates redundancy and dependency issues more rigorously than 3NF.
- ◆ Every BCNF relation is in 3NF, but the reverse is not always true.

Example:

Consider the following relation Student_Course.

Table 2.1.10 Student_Course Data

Student_ID	Course_ID	Instructor_Name
1	C1	Dr.Smith
2	C2	Dr. Johnson
3	C1	Dr.Smith

Functional Dependencies:

1. Student_ID \rightarrow Course_ID
2. Course_ID \rightarrow Instructor_Name

The above given relation Student_Course is not in BCNF because:

- ◆ Course_ID \rightarrow Instructor_Name violates BCNF because Course_ID is not a superkey, yet it determines Instructor_Name.
- ◆ The key for the table is Student_ID + Course_ID, but Course_ID alone determines Instructor_Name.

Solution to Convert into BCNF:

Decompose the table into two relations:

Table 2.1.11 Student_Course table

Student_ID	Course_ID
1	C1
2	C2
3	C1

Table 2.1.12 Course_Instructor

Course_ID	Instructor_Name
C1	Dr. Smith
C2	Dr. Johnson

Now, both tables are in BCNF.

Recap

Functional Dependency

- Functional Dependency (FD) is a concept in database design that describes a relationship between two sets of attributes in a relation (table).
- It is denoted as $X \rightarrow Y$, where:
 - X is called the determinant (a set of one or more attributes).
 - Y is called the dependent (another set of attributes).

Types of Functional Dependencies

- Partial functional dependency: a non key attribute depends on part of primary key
- Transitive functional dependency: relationship between non key attributes
- Full Functional Dependency: Non-prime attributes depend on the entire primary key, not on any subset.

Anomalies:

- Update Anomaly: Changes must be made in multiple places, risking inconsistency.
- Insertion Anomaly: Inability to insert data without violating integrity constraints.
- Deletion Anomaly: Loss of data due to deletion of related data.

Normalization: Solves these issues by decomposing tables into smaller, structured relations.

- 1NF: no multi valued dependencies
- 2NF: In 1NF, no partial dependencies
- 3NF: In 2NF, no transitive dependencies
- BCNF: In 3NF, determinants superkey

Objective Type Questions

1. Consider two sets of attributes X and Y in a relation. Let t1 and t2 are two tuples of the relation. If $t1[X] = t2[X]$, then what is the constraint on Y so that Y is functionally dependent on X?
2. Which functional dependency occurs when a non key attribute depends on part of the primary key attribute in a relation?
3. Which functional dependency occurs when there is a relationship between non key attributes in a table?
4. Consider the table below.

A	B	C
1	b1	3
2	b2	1
3	b3	3
4	b1	2

Is the functional dependency $AB \rightarrow C$ a partial, full or transitive dependency?

5. Consider the table below.

A	B	C
1	b1	3
2	b2	1
3	b3	3
4	b1	2

Will the functional dependency $C \rightarrow B$ hold true?

6. Consider $X \rightarrow Y$. Which function dependency occurs if removal of any attribute in X makes the given functional dependency invalid?
7. Consider the relation R(A, B, C, D) where AB is the primary key. Which functional dependency is satisfied by $B \rightarrow D$?
8. Consider the relation R(A, B, C, D) where AB is the primary key. Which functional dependency is satisfied by $C \rightarrow D$?
9. Which anomaly makes the database inconsistent on updating a tuple?

10. Which anomaly makes the database inconsistent on deleting a tuple?
11. Which anomaly makes the database inconsistent on inserting a tuple?
12. Which process checks whether a relation is good based on functional dependencies?
13. Which normal form satisfies the condition that the value of any attribute in a tuple must be a single value from the domain of that attribute?
14. Which normal form satisfies the condition that the relation is free from partial dependencies?
15. Which normal form satisfies the condition that the relation is free from transitive dependencies?
16. Which normal form satisfies the condition that determinants must be superkey?

Answers to Objective Type Questions

1. $t1[Y] = t2[Y]$
2. Partial functional dependency
3. Transitive functional dependency
4. Full functional dependency
5. No
6. Full functional dependency
7. Partial functional dependency
8. Transitive functional dependency
9. Updation anomaly
10. Deletion Anomaly
11. Insertion Anomaly
12. normalization
13. INF
14. 2NF

15. 3NF

16. BCNF

Assignments

1. Define functional dependency in a relational database. Explain its significance with relevant examples.
2. Explain the importance of normalization in database design. Compare and contrast 1NF, 2NF, and 3NF with appropriate examples.
3. Define BCNF and discuss how it is different from 3NF. Provide an example of converting a relation in 3NF to BCNF.
4. Discuss the concept of data redundancy and anomalies in relational databases. How does normalization help reduce redundancy and anomalies? Illustrate with examples.
5. Consider the relation $R(A, B, C, D)$ with the following functional dependencies:

$A \rightarrow B$

$B \rightarrow C$

$A \rightarrow D$

- a. Determine the candidate keys for the relation.
- b. Explain if the given relation is in 1NF, 2NF, or 3NF. Provide detailed reasoning.
- c. Suggest modifications to the relation to achieve 3NF if it is not already in that form.

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems*. McGraw-Hill.
2. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, and management* (13th ed.). Cengage Learning.
3. Foster, E., & Godbole, S. (2022). *Database systems: A pragmatic approach*. Auerbach Publications.
4. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2006). *Distributed databases. In Database system concepts* (5th ed., pp. 705-749). McGraw-Hill.
5. Groff, J. R., Weinberg, P. N., & Oppel, A. J. (2002). *SQL: The complete reference* (Vol. 2). McGraw-Hill/Osborne.

Reference

1. [geeksforgeeks.org/pl-sql-tutorial/](https://www.geeksforgeeks.org/pl-sql-tutorial/)
2. archive.nptel.ac.in/courses/106/105/106105175/
3. tutoxialspoint.com/plsql/index.htm

Unit 2

SQL Concepts

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ introduce the fundamental concepts of SQL and query execution
- ◆ explain the use of SQL commands for creating databases and tables
- ◆ identify SQL queries used for retrieving data from databases
- ◆ understand SQL queries for modifying data within a database

Prerequisites

A database organizes data in the form of tables. But how do we create a database? How can we access and modify the data within it? How do we retrieve data in a specific format? This is where a query language plays a crucial role.

A query language provides commands for creating, modifying, and retrieving data from a database, offering a standardized way to interact with it. Structured Query Language (SQL) is the most widely used query language for managing databases. It includes statements for defining data structures, modifying records, and enforcing security constraints.

Beyond basic operations such as creating, modifying, and accessing data, SQL supports advanced functionalities, including data manipulation, retrieval, and defining relationships between tables. It allows users to enforce data integrity through constraints like primary keys, foreign keys, and unique keys. Additionally, SQL enables the execution of complex queries using joins, subqueries, and aggregations, facilitating the retrieval of data in the desired format.

Due to its versatility and efficiency, SQL is an essential tool for managing relational databases and conducting data analysis. Mastering SQL is fundamental for working with modern Database Management Systems (DBMS) and is a valuable skill for database administrators, developers, and data analysts.



Keywords

Structured Query Language, SQL, Data Definition Language, DDL, Data Modification Language

Discussion

2.2.1 SQL concepts

Structured Query Language (SQL) is the most commonly used query language for databases. It consists of various statements to define and manipulate relations.

The SQL statements are categorized as follows:

Data-definition language (DDL): The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

Data-manipulation language (DML): The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.

2.2.2 SQL Data Types

SQL data type specifies the types of the attributes of a table. The basic SQL data types are:

- ◆ **Numeric:** includes integers (INTEGER or INT , and SMALLINT), floating point numbers (FLOAT or REAL , and DOUBLE PRECISION), formatted numbers. Formatted numbers can be declared by using DECIMAL (i, j) or DEC (i, j) or NUMERIC (i, j) where i, the precision, is the total number of decimal digits and j, the scale, is the number of digits after the decimal point.
- ◆ **Character-string:** includes fixed length characters or variable length characters. CHAR(n) or CHARACTER(n) indicates fixed length characters where n is the length of characters. VARCHAR (n) indicates variable length characters where n is the maximum number of characters.
- ◆ **Bit-string:** includes fixed length or varying length bits. BIT (n) indicates fixed length n number of bits. BIT VARYING (n) indicates varying length bits where n is the maximum number of bits.
- ◆ **Boolean:** data type has values of TRUE or FALSE
- ◆ **DATE:** the components of the datatype are YEAR , MONTH , and DAY in the form YYYY-MM-DD.
- ◆ **TIME:** the components of the datatype are HOUR , MINUTE , and SECOND in the form HH:MM:SS.

2.2.3 DDL Commands

2.2.3.1 CREATE DATABASE command

The CREATE DATABASE command in SQL is used to create a new database. A database is a collection of data that is stored in a structured way, and it can contain tables, views, indexes, and other database objects. This command allows you to define a new database that will hold the data for your applications.

Syntax:

```
CREATE DATABASE database_name;
```

- ◆ CREATE DATABASE is the command used to create a new database.
- ◆ database_name is the name you want to give to the database you're creating.

For example the following command creates a database with name UNIVERSITY.

```
CREATE DATABASE UNIVERSITY;
```

2.2.3.2 DROP DATABASE command

The DROP DATABASE command is used in SQL to permanently delete an entire database, including all of its tables, data, schemas, and related objects. Once executed, this action cannot be undone, and all the data stored in the database will be lost. This command is typically used when a database is no longer needed.

Syntax:

```
DROP DATABASE database_name;
```

For example, the following command deletes the database UNIVERSITY.

```
DROP DATABASE UNIVERSITY;
```

2.2.3.3 CREATE TABLE Command

CREATE TABLE command creates a table in a database. We should specify the attributes and initial constraints of the table in the CREATE TABLE command.

Syntax:

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

- ◆ **table_name:** The name of the table you want to create.
- ◆ **column1, column2, columnN:** The names of the columns within the table
- ◆ **data type:** The data type for each column, specifying the type of data that can be stored in it.
- ◆ **constraints:** optional constraints, such as primary keys, unique constraints, and foreign keys, that define rules and relationships within the table.

Constraints in DBMS Table Creation

Constraints in Database Management Systems (DBMS) ensure the accuracy and reliability of data by enforcing rules on the values that can be inserted, updated, or deleted. They help maintain data integrity and prevent invalid data entry. Common constraints include Primary Key, Foreign Key, Unique, Not Null, Check, and Default.

Types of Constraints

When creating a table in SQL, constraints can be applied at the column level (to a specific column) or at the table level (affecting multiple columns). Below are the main types of constraints:

1. NOT NULL Constraint

Ensures that a column cannot contain NULL values. It is used when a field must always have data, such as a primary key or an email. For example, an employee ID and name cannot be NULL because every employee must have a unique ID and a name.

2. UNIQUE Constraint

Ensures that all values in a column are distinct, preventing duplicate entries while allowing NULLs (unless the NOT NULL constraint is also applied). For example, each employee's ID must be unique to distinguish them from others.

3. PRIMARY KEY Constraint

Uniquely identifies each record in a table. Combines NOT NULL + UNIQUE constraints. Only one primary key per table, but it can have multiple columns (composite primary key). EmpID uniquely identifies each employee.

4. FOREIGN KEY Constraint

Enforces referential integrity by linking a column to another table's **Primary Key**. Prevents actions that would destroy links between tables. **CustomerID** in **Orders** must match an existing **CustomerID** in **Customers**.

5. CHECK Constraint

Ensures that values in a column meet a specific condition. Used for data validation. **Age** must be at least 18, and **Salary** must be greater than 0.

6. DEFAULT Constraint

Assigns a default value if no value is provided during insertion. If no value is inserted for **Status**, it defaults to 'Active'.

7. AUTO_INCREMENT (MySQL only)

Automatically generates unique values for a column. Commonly used for primary keys. **EmpID** automatically increments for new records.

Summary Table

Constraint	Description	Example
NOT NULL	Ensures a column cannot be NULL	Age INT NOT NULL
UNIQUE	Ensures all values are distinct	Email VARCHAR(100) UNIQUE
PRIMARY KEY	Uniquely identifies a row (NOT NULL + UNIQUE)	EmpID INT PRIMARY KEY
FOREIGN KEY	Ensures referential integrity	FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
CHECK	Restricts values based on a condition	CHECK (Age >= 18)
DEFAULT	Assigns a default value if none is provided	Status VARCHAR(20) DEFAULT 'Active'
AUTO_INCREMENT	Auto-generates unique values (MySQL)	EmpID INT AUTO_INCREMENT PRIMARY KEY

For example, the following command creates a table STUDENT with ID as the primary key. NOT NULL is a constraint that specifies that the value of the field cannot be null.

Example:

```
CREATE TABLE STUDENT(  
  ID INT NOT NULL,  
  NAME VARCHAR(20) NOT NULL,  
  SEMESTER INT NOT NULL,  
  DEPARTMENT CHAR(25),  
  PRIMARY KEY (ID)  
);
```

2.2.3.4 DROP TABLE

The DROP TABLE command in SQL is used to permanently delete a table from a database. When executed, it removes the specified table, including all its data, structure, and associated objects such as indexes or constraints.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP Table STUDENT;
```

The above command will completely remove the STUDENT table from the database. Once a table is dropped, all its data is lost, and this action cannot be undone unless a backup exists.

2.2.4 Selecting and checking databases

Checking Databases: Involves listing all available databases in the system using commands like SHOW DATABASES; to view and verify which databases exist.

Syntax:

```
SHOW DATABASES;
```

Example Output:

After executing SHOW DATABASES;, you might see output like the table 2.2.1

Table 2.2.1 Result of SHOW DATABASE command

Databases
UNIVERSITY_Database
EMPLOYEE_Database
STUDENT_Database

Selecting a Database:

Once you identify the database you want to use, you can make it active with the USE command.

Syntax:

```
USE database_name;
```

Example:

```
USE UNIVERSITY;
```


This sets UNIVERSITY as the active database.

2.2.5 DML Commands (DATA MANIPULATION LANGUAGE)

2.2.5.1 INSERT INTO Command

Add rows into a table

The INSERT INTO command in SQL is used to add rows to a table. To insert a row, you must specify the table name and provide a list of values. The values should be listed in the same order as the attributes defined in the CREATE TABLE command. For example, the following command adds a row to the STUDENT table:

```
INSERT INTO STUDENT  
VALUES (1, 'Aman', 2, 'BCA');
```

Another form of the INSERT command allows you to explicitly specify the attribute names. This is useful when you want to insert values for only a subset of attributes in the table. However, you must ensure that all attributes with NOT NULL constraints and no default values are included. For example:

```
INSERT INTO STUDENT(ID, NAME, SEMESTER)  
VALUES (1, 'Aman', 2);
```

2.2.5.2 SELECT command

The SELECT statement in SQL is used to retrieve information from databases. It is one of the most commonly used commands for querying data. The basic structure of a SELECT statement is as follows:

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;
```

where

- ◆ attribute list is a list of attribute names whose values are to be retrieved by the query.
- ◆ table list is a list of the relation names required to process the query.
- ◆ condition is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic comparison operators are =, <, <=, >, >= and <> which corresponds to the relational algebra operations =, ≤, ≤, ≥, ≥, and ≠.

Table 2.2.2 STUDENT table with attributes ID, NAME, SEMESTER, and DEPARTMENT

ID	NAME	SEMESTER	DEPARTMENT
1	RAM	1	BCA
2	GEORGE	2	HISTORY
3	KEERTI	3	ENGLISH
4	IRFAN	1	ECONOMICS
5	KOMAL	2	MALAYALAM
6	KIRAN	4	MATHEMATICS
7	NIRMAL	1	ENGLISH

Consider the Table 2.2.2, to retrieve the ID and NAME of the students in semester 1 we have to give the query:

```
SELECT ID, NAME
FROM STUDENT
WHERE SEMESTER=1;
```

The result of the above query will be displayed as shown in Table 2.2.3.

Table 2.2.3 Result of SELECT command

ID	NAME
1	RAM
4	IRFAN
7	NIRMAL

If you want to retrieve all the rows of STUDENT table use the following query:

```
SELECT * FROM STUDENT;
```

AND and OR operators can be used to combine multiple conditions in the WHERE clause. For example, the following query will retrieve the ID and NAME of students in first semester BCA.

```
SELECT ID, NAME
FROM STUDENT
WHERE SEMESTER=1 AND DEPARTMENT='BCA';
```

Result of above query is shown in table 2.2.4

Table 2.2.4 SELECT command with AND and OR operators

ID	NAME
1	RAM

2.2.5.3 DELETE command

The DELETE command in SQL is used to remove one or more rows from a table based on a specified condition. It allows you to delete data without affecting the structure of the table. For example, the following query on table 2.2.2 will produce the result in table 2.2.5

```
DELETE FROM STUDENT
```

```
WHERE ID=3;
```

Result of the above query is:

Table 2.2.5 Result of DELETE query after deleting the record with ID=3

ID	NAME	SEMESTER	DEPARTMENT
1	RAM	1	BCA
2	GEORGE	2	HISTORY
4	IRFAN	1	ECONOMICS
5	KOMAL	2	MALAYALAM
6	KIRAN	4	MATHEMATICS
7	NIRMAL	1	ENGLISH

2.2.5.4 UPDATE command

The UPDATE command is used to modify rows in a table.

Syntax:

```
UPDATE table_name  
  
SET column1 = value1, column2 = value2, ...  
  
WHERE condition;
```

The SET clause in the UPDATE command specifies the attributes to be modified and their new values. For example, the following query will change the semester of the student with ID 2 to 3 in the table 2.2.2. The result is shown in table 2.2.6.

```
UPDATE STUDENT
```

```
SET SEMESTER=3
```

```
WHERE ID=2;
```

Result of the above query is:

Table 2.2.6 Result of UPDATE query after updating the semester of student with ID=3

ID	NAME	SEMESTER	DEPARTMENT
1	RAM	1	BCA
2	GEORGE	3	HISTORY
3	KEERTI	3	ENGLISH
4	IRFAN	1	ECONOMICS
5	KOMAL	2	MALAYALAM
6	KIRAN	4	MATHEMATICS
7	NIRMAL	1	ENGLISH

2.2.6 Substring pattern matching

In SQL, substring pattern matching can be performed using several string functions. The most common ones are:

2.2.6.1 LIKE Operator

The LIKE operator is used to search for a specified pattern in a column. It is typically used with wildcard characters (%) and (_) for pattern matching.

_: Represents zero or more characters.

_: Represents exactly one character.

Syntax:

```
SELECT column_name
FROM table_name
WHERE column_name LIKE 'pattern';
```

Example:

The following query retrieves the details of all the students whose name starts with 'K'.

```
SELECT * FROM STUDENT
WHERE NAME LIKE 'K%';
```

Result is shown in table 2.2.7

Table 2.2.7 Result of LIKE operator

ID	NAME	SEMESTER	DEPARTMENT
3	KEERTI	3	ENGLISH
5	KOMAL	2	MALAYALAM
6	KIRAN	4	MATHEMATICS

2.2.7 Ordering of query result

ORDER BY clause can be used to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result. For example, the following query retrieves the ID and NAME of all students in the STUDENT table with names ordered alphabetically in ascending order. ASC or DESC keyword is used with order by clause to retrieve result in ascending or descending order.

If not specify anything it will generate ascending order of selected column.

For example,

```
SELECT ID, NAME
FROM STUDENT
ORDER BY NAME;
```

Result:

Table 2.2.8 Result of ORDER BY command

ID	NAME
2	GEORGE
4	IRFAN
3	KEERTI
6	KIRAN
5	KOMAL
7	NIRMAL
1	RAM

If we have to retrieve the results in the descending order of names, use the DESC keyword.

```
SELECT ID, NAME
FROM STUDENT
ORDER BY NAME DESC;
```

Result:

Table 2.2.9 Result of ORDER BY command with DESC Keyword

ID	NAME
1	RAM
7	NIRMAL
5	KOMAL
6	KIRAN

3	KEERTI
4	IRFAN
2	GEORGE

2.2.8 Aggregate functions

Aggregate functions are used to perform calculations on a set of values and return a single result. The five built-in aggregate functions:

8. Average: AVG
9. Minimum: MIN
10. Maximum: MAX
11. Total: SUM
12. Count: COUNT

2.2.8.1 Average: AVG

The AVG function calculates the average (mean) of a set of numeric values. It is commonly used to find the average salary, grade, or any other numeric metric in a table.

Syntax:

```
SELECT AVG(column_name)
FROM table_name;
```

Example:

Consider the following table 2.2.10

Table 2.2.10 STUDENT table with attributes ID, NAME, SEMESTER, MARKS and DE- PARTMENT

ID	NAME	SEMESTER	MARKS	DEPARTMENT
1	RAM	1	90	BCA
2	GEORGE	3	95	HISTORY
3	KEERTI	3	90	ENGLISH
4	IRFAN	1	85	ECONOMICS
5	KOMAL	2	75	MALAYALAM
6	KIRAN	4	60	MATHEMATICS
7	NIRMAL	1	72	ENGLISH

The AVG function calculates the average marks obtained by all students in the given table. Consider the following query:

```
SELECT AVG(MARKS)
```

FROM STUDENT

WHERE DEPARTMENT='ENGLISH';

The query calculates the average marks of students who belong to the 'ENGLISH' department. It uses the AVG() function to compute the mean value of the MARKS column, filtering only those rows where the DEPARTMENT is 'ENGLISH'.

Result:

AVG(MARKS): 81

2.2.8.2 Minimum: MIN

The MIN function returns the smallest value in a column. It is used to find the minimum value in datasets like the lowest price, minimum marks, or shortest time.

Syntax:

SELECT MIN(column_name) FROM table_name;

Example:

Consider the following query

SELECT MIN(MARKS)

FROM STUDENT

WHERE DEPARTMENT = 'ENGLISH';

The above query retrieves the minimum marks of students from the ENGLISH department by filtering rows with DEPARTMENT = 'ENGLISH' and applying the MIN() function on the MARKS column.

Result:

MIN(MARKS): 72

2.2.8.3. Maximum (MAX)

The MAX function retrieves the largest value in a column. For example It helps identify the highest salary, maximum marks, or largest order value in a dataset.

Syntax:

SELECT MAX(column_name) FROM table_name;

Example:

Consider the following query

```
SELECT MAX(MARKS)
FROM STUDENT
WHERE DEPARTMENT = 'ENGLISH';
```

The above SQL query retrieves the highest marks (maximum value) from the MARKS column in the STUDENT table, but only for rows where the DEPARTMENT is 'ENGLISH'.

Result:

MAX(MARKS): 90

2.2.8.4 Total (SUM)

The SUM function adds all the numeric values in a column and returns the total.

Syntax:

```
SELECT SUM(column_name)
FROM table_name;
```

Example:

Consider the following query:

```
SELECT SUM(MARKS)
FROM STUDENT
WHERE DEPARTMENT = 'ENGLISH';
```

The above query calculates the total marks of all students in the ENGLISH department by summing up the values in the MARKS column.

Result:

SUM(MARKS): 162

2.2.9 Aggregation with grouping

The GROUP BY clause is used to group rows (tuples) in a table based on the values of one or more specified columns. Tuples with the same value in all the columns mentioned in the GROUP BY clause are combined into one group, and aggregate functions (like AVG, SUM, COUNT, etc.) are applied to each group. For example, to find the average marks of students in each department, the following query can be used:

Consider the following STUDENT table 2.2.11

Table 2.2.11 STUDENT table with attributes ID, NAME, SEMESTER, MARKS, DEPARTMENT

ID	NAME	SEMESTER	MARKS	DEPARTMENT
1	RAM	1	90	BCA
2	GEORGE	3	95	HISTORY
3	KEERTI	3	90	ENGLISH
4	IRFAN	1	85	ECONOMICS
5	KOMAL	2	75	MALAYALAM
6	KIRAN	4	60	MATHEMATICS
7	NIRMAL	1	72	ENGLISH

Consider the following query:

```
SELECT DEPARTMENT, AVG(MARKS)
FROM STUDENT
GROUP BY DEPARTMENT;
```

This query calculates the average marks of students in each department using the AVG() function. The GROUP BY DEPARTMENT clause groups the rows by department, and the average of the MARKS column is calculated for each group.

Result:

Table 2.2.12 Result of aggregation with grouping

DEPARTMENT	AVG(MARKS)
BCA	90
HISTORY	95
ENGLISH	81
ECONOMICS	85
MALAYALM	75
MATHEMATICS	60

2.2.10 HAVING clause

The HAVING clause is used to specify conditions on groups created by the GROUP BY clause, typically in combination with aggregate functions like AVG, SUM, or COUNT. Unlike the WHERE clause, which filters rows before grouping, the HAVING clause filters groups after the aggregation. For example, to find the departments with an average mark greater than 90, use the following query:

```
SELECT DEPARTMENT, AVG(MARKS)
FROM STUDENT
GROUP BY DEPARTMENT
HAVING AVG(MARKS)>90;
```

Result:

Table 2.2.13 Result of Having clause

DEPARTMENT	AVG(MARKS)
HISTORY	95

Recap

- ◆ SQL: Most widely used query language
- ◆ SQL DDL: Commands for defining relation schemas, deleting relations, and modifying relation schemas.
- ◆ SQL DML: Query information, insert tuples, delete tuples from, and modify tuples in the database.
- ◆ SQL data type: Types of the attributes
- ◆ Basic SQL datatypes: Numeric, Character string, Bit string, Boolean, DATE, and TIME
- ◆ CREATE DATABASE command: Creates a new database.
- ◆ DROP DATABASE command: Deletes an existing database.
- ◆ SHOW DATABASES: To check available databases
- ◆ CREATE TABLE command: Creates a table in a database.
- ◆ DROP TABLE command: Delete a table in a database.
- ◆ INSERT: Add rows in a table
- ◆ SELECT: Retrieve information from databases.
- ◆ DELETE: Remove rows in a table.
- ◆ UPDATE: Modify rows in a table.
- ◆ The LIKE comparison operator: String pattern matching.
- ◆ ORDER BY clause: To order the tuples in the result of a query
- ◆ Built-in aggregate functions: AVG, MIN, MAX, SUM, COUNT
- ◆ GROUP BY clause: Group a set of tuples.
- ◆ HAVING clause: Specify conditions to groups.

Objective Type Questions

1. Which SQL clause is used to specify conditions to groups?
2. Which SQL command is used for deleting a table in a database?
3. Which SQL command is used for modifying the rows in a table?
4. Which SQL clause is used to order the tuples in the result of a query?
5. Which SQL command is used to add rows into a table?
6. Which SQL command is used for retrieving information from databases?
7. Which SQL clause is used to group a set of tuples?
8. Which SQL command creates a table in a database?
9. Which SQL command checks available databases?
10. Which SQL command is used for removing rows in a table?
11. Which SQL command creates a new database?
12. Which SQL comparison operator is used for string pattern matching?
13. Which SQL command selects a database from available databases?
14. Which SQL command deletes an existing database?

Answers to Objective Type Questions

1. HAVING
2. DROP TABLE
3. UPDATE
4. ORDER BY
5. INSERT INTO
6. SELECT
7. GROUP BY
8. CREATE TABLE
9. SHOW DATABASES
10. DELETE
11. CREATE DATABASE
12. LIKE
13. USE
14. DROP DATABASE

Assignments

1. Explain the difference between DDL (Data Definition Language) and DML (Data Manipulation Language) commands in SQL. Use the EMPLOYEE table below to perform the following:

- Create the table using CREATE TABLE.
- Insert 3 rows into the table using INSERT.
- Delete one record using DELETE.

ID	NAME	DEPARTMENT	SALARY
1	HELEN	HR	50000
2	JAMES	IT	70000
3	ALICE	FINANCE	60000

2. Describe various SQL data types with examples. Create a table called COURSES using appropriate data types to store the following data, and explain the structure:
3. Explain DML & DDL commands.
4. Explain different constraints database creation.

Suggested Reading

1. Teate, Renee MP. *SQL for Data Scientists: A Beginner's Guide for Building Datasets for Analysis*. John Wiley & Sons, 2021.
2. Groff, James R., Paul N. Weinberg, and Andrew J. Oppel. *SQL: the complete reference. Vol. 2*. McGraw-Hill/Osborne, 2002.
3. Beaulieu, Alan. *Learning SQL: master SQL fundamentals*. O'Reilly Media, 2009.
4. CELKO, J., and S. Q. L. Joe Celko's. *"for Smarties: Advanced SQL Programming*. 3. vyd. Massachusetts." (2005).

Reference

1. [geeksforgeeks.org/pl-sql-tutorial/](https://www.geeksforgeeks.org/pl-sql-tutorial/)
2. archive.nptel.ac.in/courses/106/105/106105175/
3. tutoxialspoint.com/plsql/index.htm

SGOU

Unit 3

Built-in Functions

Learning Outcomes

The learner will be able to:

- ◆ define what built-in functions are in a programming language
- ◆ identify commonly used aggregate functions
- ◆ recall the purpose of numeric functions
- ◆ recognize string functions in a given code snippet

Prerequisites

Before learning about built-in functions, you may already know how to write basic programs that take input, process data, and display output. You might have used variables to store values and operators to perform calculations. These fundamental concepts help in solving simple problems using code.

However, as programs become more complex, writing code for every small task can take a lot of time. For example, if you want to find the length of a word or calculate the square root of a number, writing the logic manually would be difficult and time-consuming. Similarly, tasks like converting text to uppercase, finding the largest number in a list, or calculating the sum of multiple values would require additional coding effort. Programming languages provide built-in functions, which are ready-made functions designed to perform common tasks quickly and efficiently.

Built-in functions not only save time but also reduce the chances of errors in your code. Instead of writing multiple lines of logic, you can achieve the same results with just a single function call. This makes programming more efficient, readable, and easier to maintain.

In this lesson, you will learn about different types of built-in functions, including scalar functions, aggregate functions, numeric functions, and string functions. Understanding these will help you write programs more easily, making your code faster, simpler, and more accurate.



Keywords

Numeric functions, String functions, CONCAT(), Scalar functions

Discussion

2.3.1 Introduction to Built-in Functions in SQL

Built-in functions in SQL are pre-defined functions provided by the database management system to perform operations on data. These functions help simplify common tasks like mathematical calculations, string manipulation, date and time handling, and more.

They can be used in SELECT, WHERE, ORDER BY, and other SQL clauses to process and transform data easily and efficiently.

Features of Built in Function are

1. **Predefined and Ready to Use:** Built-in functions are provided by the SQL language or database system, requiring no user definition or setup.
2. **Simplifies Complex Operations:** Functions like SUM(), LEN(), GETDATE() make complex operations easy with just a single call.
3. **Improves Query Efficiency:** They reduce the need for lengthy and repetitive SQL code, making queries faster and more maintainable.
4. **Supports Multiple Data Types:** Built-in functions can work on strings, numbers, dates, and more.
5. **Can Be Used in Various Clauses:** These functions can be used in SELECT, WHERE, GROUP BY, ORDER BY, and HAVING clauses.

2.3.1.1 Importance of built in functions for data Manipulation and Querying

Built-in functions are incredibly valuable tools for working with data. They provide ready-made solutions for handling common tasks like cleaning, transforming, and analyzing data, which can save a lot of time and effort. For example, in Python, functions like len(), sum(), and min() allow you to quickly calculate the length of a list, the total of numbers, or the smallest value without having to write extra code. Similarly, in databases like SQL, functions such as COUNT(), AVG(), and GROUP BY make it simple to summarize and organize data during queries.

Using built-in functions also makes your code easier to understand and maintain. Since these functions are well-documented and widely used, others who read your code can easily follow what you have done. For instance, Python's pandas library includes tools like merge() for combining data, groupby() for aggregating it, and pivot() for reshaping it—all of which make data manipulation more straightforward. These functions allow you to perform complex operations in a clear and concise way, which is

especially useful when analyzing data or creating reports.

Another major benefit of built-in functions is that they are optimized for performance. Instead of writing your own algorithms, you can rely on these pre-designed tools, which are often faster and more efficient. For example, SQL functions like JOIN and WHERE help you filter or combine data in ways that are both powerful and quick. Similarly, libraries like NumPy in Python include vectorized functions such as mean() and std() that can process large datasets much faster than traditional loops. This efficiency is particularly important when working with big data, where speed can make a huge difference.

2.3.2 Scalar Functions

Scalar functions in DBMS are predefined functions that operate on a single value (one row at a time) and return one single value as a result. They are used for string manipulation, mathematical operations, date processing, and type conversion.

Table 2.3.1 Example Database: STUDENT

Student ID	Name	Marks	City	DOB
1	John	85	New York	2000-05-14
2	Alice	92	Chicago	1999-11-22
3	Bob	76	Boston	2001-03-10
4	Carol	89	Dallas	2000-09-05

Types of Scalar Functions:

2.3.2.1 String Functions

String functions in DBMS are built-in functions used to manipulate and perform operations on string (text) data. They help in modifying, analyzing, and formatting string values in a database.

- a. UPPER() → Converts text to uppercase

```
SELECT UPPER (Name) FROM STUDENT;
```

Output:

UPPER(Name)
JOHN
ALICE
BOB
CAROL

b. LOWER() → Converts text to lowercase

SELECT LOWER(Name) From Student;

Output:

LOWER (Name)
JOHN
ALICE
BOB
CAROL

c. LENGTH() → Returns length of string

SELECT Name, LENGTH(Name) FROM STUDENT;

Output:

Name	LENGTH(Name)
John	4
Alice	5
Bob	3
Carol	5

d.CONCAT() → Combines two strings

SELECT CONCAT(Name, ' from ', City) AS Student_Info FROM STUDENT;

Output:

Student_Info
John from New York
Alice from Chicago
Bob from Boston
Carol from Dallas

2.3.2.2 Numeric Functions

Numeric functions are built-in functions in DBMS used to perform mathematical operations on numeric data types like integers, decimals, and floats. These functions help in calculations, rounding, and other number-related tasks within SQL queries.

a. ROUND() → Rounds numeric value

```
SELECT Marks, ROUND(Marks/7, 2) AS GPA FROM STUDENT;
```

Output:

Marks	GPA
85	12.14
92	13.14
76	10.86
89	12.71

b. MOD() → Finds remainder

```
SELECT Marks, MOD(Marks, 2) AS Even_Odd FROM STUDENT;
```

Output:

Marks	Even_Odd
85	1
92	0
76	0
89	1

2.3.2.3 Date Functions

a. YEAR() → Extracts year from date

```
SELECT Name, YEAR(DOB) AS Birth_Year FROM STUDENT;
```

Output:

Name	Birth_Year
John	2000
Alice	1999
Bob	2001
Carol	2000

b. CURDATE() → return the current data in YYYY-MM-DD Format

```
SELECT CURDATE() As Today_Date;
```

Today_Date
2025-04-03

c. DATEDIFF() → Difference between two dates

```
SELECT Name, DATEDIFF(CURDATE(), DOB) AS Age_Days FROM STUDENT;
```

Name	Age_Days
John	9220
Alice	9585
Bob	8855
Carol	9220

2.3.2.4 Conversion Functions

a. CAST() → Converts data type

```
SELECT CAST(Marks AS CHAR) FROM STUDENT;
```

Converts numeric Marks to string.

Output:

Marks
'85'
'92'
'76'
'89'

The values are now stored as text, not numbers.

2.3.3 Aggregate Functions

Aggregate functions in DBMS are **predefined functions** that perform **calculation or summarization on a group of rows (multiple values)** and return a **single result**. They are mostly used with **GROUP BY** clause in SQL to group data and perform operations on each group.

Table 2.3.2 Example Database: EMPLOYEE

Emp ID	Name	Department	Salary	Age
1	John	HR	50000	30
2	Alice	IT	70000	28
3	Bob	IT	60000	35
4	Carol	HR	55000	40
5	David	Finance	75000	45

Common Aggregate Functions:

1. **COUNT()** → Counts number of rows

```
SELECT COUNT(*) FROM EMPLOYEE;
```

Output: 5 (Total number of employees)

2. **SUM()** → Adds up values in a column

```
SELECT SUM(Salary) FROM EMPLOYEE;
```

Output: 310000 (Total salary: 50000+70000+60000+55000+75000)

3. **AVG()** → Calculates average value

```
SELECT AVG(Salary) FROM EMPLOYEE;
```

Output: 62000 (Total salary / Number of employees = 310000 / 5)

4. **MIN()** → Finds minimum value

```
SELECT MIN(Age) FROM EMPLOYEE;
```

Output: 28 (Youngest employee's age)

5. **MAX()** → Finds maximum value

```
SELECT MAX(Salary) FROM EMPLOYEE;
```

Output: 75000 (Highest salary)

2.3.3.1 Using Aggregate Functions with GROUP BY

We can also group data and apply aggregate functions per group.

Example: Calculate total salary per department:

SELECT Department, SUM(Salary)

FROM EMPLOYEE

GROUP BY Department;

Output:

Department	SUM(Salary)
HR	105000
IT	130000
Finance	75000

Recap

- ◆ Built-in functions in SQL help simplify common data tasks and save time.
- ◆ These functions are already available in SQL, so no need to write complex code.
- ◆ The COUNT() function is used to determine how many rows are in a table.
- ◆ COUNT() can also be used to count specific records that meet certain conditions.
- ◆ The SUM() function adds the values in a numeric column.
- ◆ SUM() is useful for calculating totals, like total sales or expenses.
- ◆ The AVG() function calculates the average value of a numeric column.
- ◆ AVG() helps determine things like the average salary or score.
- ◆ Built-in functions are predefined functions that perform common operations in SQL.
- ◆ Scalar functions return a single value based on the input provided.
- ◆ String functions help manipulate text, such as CONCAT for joining strings, LENGTH for finding text length and UPPER for converting text to uppercase.
- ◆ Numeric functions are used to carry out operations on numerical values, such as ROUND, which rounds numbers, and MOD, which calculates the remainder of a division

- ◆ Date functions handle date and time data, like YEAR, which retrieves the year portion from a date value.
- ◆ Conversion functions allow data type conversions, like CAST.
- ◆ Aggregate functions perform calculations on multiple rows of data to return a single result.
- ◆ Common aggregate functions include SUM, AVG, COUNT, MIN, and MAX.
- ◆ Aggregate functions with GROUP BY help analyze and summarize data by grouping related records.

Objective Type Questions

1. What does the LENGTH() function do in SQL?
2. Which function would you use to convert all letters in a string to lowercase?
3. Which SQL function is used to compute the difference between two date values?
4. Give an example of a function that summarizes data across multiple rows.
5. What SQL function returns the total number of rows in a result set?
6. Which function helps determine the largest value within a column?
7. Which function can be used to return the current date?
8. Which function would you use to convert all letters in a string to lowercase?
9. Which SQL function helps retrieve the smallest value from a column?
10. What SQL function would you use to change the data type of a value?

Answers to Objective Type Questions

1. It returns the number of characters in a string.
2. LOWER()
3. DATEDIFF()
4. SUM()
5. COUNT()
6. MAX()

7. CURRENT_DATE
8. LOWER()
9. MIN()
10. CAST()

Assignments

1. Explain the concept of aggregate functions in SQL. Write queries using at least three aggregate functions and explain their outputs using a sample table.
2. Write a query that displays the number of days each student has lived using the DOB column and the current date. Describe how the DATEDIFF() function works in this scenario.
3. Define scalar functions in SQL. How are they different from aggregate functions? Write examples using scalar functions with an explanation.
4. Describe how numeric functions are used in SQL to perform mathematical operations. Write queries and explain the results with sample values.
5. What are string functions in SQL? Write a query using UPPER(), LOWER(), and CONCAT() functions and describe how each function transforms the data.

Suggested Reading

1. Teate, R. M. P. (2021). *SQL for data scientists: A beginner's guide for building datasets for analysis*. John Wiley & Sons.
2. Groff, J. R., Weinberg, P. N., & Oppel, A. J. (2002). *SQL: The complete reference* (Vol. 2). McGraw-Hill/Osborne.
3. Beaulieu, A. (2009). *Learning SQL: Master SQL fundamentals*. O'Reilly Media.
4. Celko, J. (2005). *Joe Celko's SQL for smarties: Advanced SQL programming* (3rd ed.). Morgan Kaufmann.

Reference

1. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage Learning.
2. Pratt, P. J., & Last, M. Z. (2020). *Concepts of database management* (10th ed.). Cengage Learning.
3. Oppel, A. J. (2015). *SQL: A beginner's guide* (4th ed.). McGraw-Hill Education
4. Libeskind, R. (2020). *Understanding SQL: A beginner's guide* (1st ed.). Independently published.
5. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill Education.

Unit 4

Views and Transaction Control Commands

Learning Outcomes

The learner will be able to:

- ◆ define what a view is in SQL
- ◆ identify the purpose of Transaction Control Commands (TCC) in SQL
- ◆ list the different types of Transaction Control Commands
- ◆ recall the syntax for creating a view in SQL

Prerequisites

Before learning about Views and Transaction Control Commands, you may already be familiar with writing basic SQL queries to retrieve, insert, update, and delete data from tables. You might have used `SELECT` statements to fetch records and `WHERE` clauses to filter data. These skills help in managing and analyzing information stored in a database.

However, as databases grow larger and more complex, managing data efficiently becomes essential. Imagine having to write long and repetitive queries every time you need specific information. This is where Views come in—they allow you to save and reuse queries as virtual tables, making data retrieval easier and more organized.

Similarly, when working with databases, mistakes can happen. What if you accidentally delete important records or update the wrong data? Transaction Control Commands (TCC) help maintain data accuracy by allowing you to commit changes, undo mistakes, or save checkpoints before making permanent modifications.

In this lesson, you will learn how Views simplify database management and how Transaction Control Commands ensure data integrity. These concepts will help you work more efficiently and prevent errors while handling large amounts of data.

Keywords

Commit, Rollback, Savepoint, Atomicity, Durability



Discussion

2.4.1 Introduction to Views

A view in a database is like a window that shows a specific part of the data from one or more tables. It does not actually store data itself but instead holds a saved query that gets the data when needed. Think of it as a custom way to look at the data without changing the original tables. Views make it easier to work with complex data by simplifying how we access it, and they can also protect sensitive information by allowing access to only certain parts of the data.

To create a view, we use a special SQL command that tells the database what kind of data to show. Views are useful because they help organize and present data in a way that fits a specific need, without altering how the data is stored. There are different types of views: simple views that pull data from a single table, and complex views that combine information from multiple tables. Some views allow us to make changes to the data they show, while others only allow viewing without editing.

2.4.1.1 Purpose of Views in SQL

A view in SQL is like a virtual table that combines data from one or more real tables. It is created using a simple query, and it serves several important purposes.

1. **Simplifies Complex Queries:** Views make it easier to work with complicated queries. Instead of writing long and complex SQL commands every time, you can just create a view once and use it whenever needed.
2. **Improves Security:** Views help protect sensitive data. For example, if you do not want certain users to see all the information in a table, you can create a view that only shows the necessary data, keeping other data hidden.
3. **Provides Abstraction:** Views allow users to access data without knowing the details of the database structure. This makes it easier to work with the data because users do not need to understand the complexity behind it.
4. **Makes Data Reusable:** Once a view is created, it can be used in multiple places. This saves time and effort since you do not have to write the same query repeatedly.

2.4.1.2 How Views are Different from Tables

A table is like a storage box in a database where data is kept in rows and columns. It holds actual information, such as names, numbers, or dates. You can change or add new data directly in a table.

A view, however, is like a window that shows data from one or more tables. It does not store the data itself but simply displays it based on a query (or question) you ask the database. For example, a view might show only certain columns from a table or only specific rows of data. Unlike tables, you can not directly change data in a view, but you can see it as if it were a table. Views are useful for simplifying complex information or protecting sensitive data by showing only what is necessary.

In short:

- ◆ Table: Stores real data and allows changes.
- ◆ View: Shows data from tables, but doesn't store it and is mostly for viewing.

2.4.2 Creating Views

A view is a SQL query that is permanently stored in a database and assigned a name. To the database user, a view looks like a real table. But the view does not exist in a database. The rows and columns in a view are the results of the query that defines the view.

2.4.2.1 Syntax for Creating a View

To define a view, we must give the view a name and must state the query that computes the view. The form of the create view command is:

```
create view v as <query expression>;
```

where v is the view name and query expression is any logical query.

2.4.2.2 Example of Creating a Simple View

Consider a student who wants to know the courses taken by a faculty but not authorized to access the salary details of the faculty in an INSTRUCTOR table. A view named FACULTY can be given to the student.

```
CREATE VIEW FACULTY AS
```

```
SELECT ID, NAME, DEPARTMENT, COURSES FROM INSTRUCTOR;
```

2.4.2.3 Using Views in SQL

After defining the view, the view name can be used to refer to the relation generated by the view. Suppose, we need to find the DEPARTMENT of the faculties in the FACULTY view, use the following query.

```
SELECT NAME,  
DEPARTMENT FROM  
FACULTY;
```

The relation generated by a view is the result of the evaluation of the query that defines the view. If the relation defining a view is modified, then the view becomes out of date. To prevent this, views are implemented as follows. When we define a view, the database stores the definition of the view. When a view relation is used in a query, it is replaced by the query expression. Whenever the query is executed, the view is recomputed.

2.4.3 Update of a View

If the database is expressed in terms of a view, then the modifications to it might cause some problems. All the insertions, deletions, and updates in a query using view should be translated to a modification in the actual table.

An SQL view is said to be updatable (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- ◆ The from clause has only one database relation.
- ◆ The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
- ◆ Any attribute not listed in the select clause can be set to null; that is, it does not have a not null constraint and is not part of a primary key.
- ◆ The query does not have a group by or having clause.

Suppose we want to update the department of the faculty named TOM to ENGLISH in the FACULTY view use the following query. The query satisfies all the above conditions, therefore the the view will be updated and the update will be reflected in the actual table i.e. INSTRUCTOR.

```
UPDATE FACULTY
```

```
SET DEPARTMENT="ENGLISH" WHERE NAME = "TOM";
```

2.4.3.1 Inserting and Deleting rows in a view

The rules for updating the view applies to INSERT and DELETE operations on a view. For example to delete a row in FACULTY view with ID=3 use the following query.

```
DELETE FROM FACULTY WHERE ID=3;
```

This query will delete a row from the actual INSTRUCTOR table and it will be reflected in the view FACULTY.

WITH CHECK OPTION

WITH CHECK OPTION at the end of the view definition ensures that the insertions, deletions, and updates satisfies the WHERE clause in the view definition. If a tuple inserted the WHERE condition of the view with WITH CHECK option, then the insertion is rejected. Updates and delete are similarly rejected if they do not satisfies the WHERE clause conditions in view. The following query is an example of a view with WITH CHECK option.

```
CREATE VIEW FACULTY AS SELECT ID, NAME, DEPARTMENT FROM  
INSTRUCTORS
```

```
WHERE DEPARTMENT IS NOT NULL WITH CHECK OPTION;
```

The WITH CHECK option in the above query denies the entry of NULL values in DEPARTMENT column of the view.

2.4.4 Types of Views

In SQL, a view is a virtual table created by querying one or more base tables. It can simplify data retrieval by presenting data in a specific format or by hiding complex queries. There are different types of views, each serving a unique purpose depending on the complexity of the data being queried.

◆ Simple view

A simple view is a view based on a single table. It typically selects specific columns or rows from the table, providing a simplified version of the data. For example, if a database contains an employee table, a simple view might display only the employee names and salaries, excluding other details like employee ID or department. Simple views are easy to create and use, offering an efficient way to extract relevant data from one table.

◆ Complex view

A complex view, on the other hand, involves more than one table and usually includes joins. These views allow users to combine data from different tables, providing a more comprehensive view of related information. For instance, a complex view might join an employee table with a department table to display employee names along with their department names. Complex views can also include filtering, grouping, and aggregation to provide more detailed insights.

◆ Materialized view

Materialized views are a special type of view that stores the result of a query physically, rather than computing it every time the view is accessed. These views are particularly useful when dealing with large datasets, as they can improve performance by reducing the need to recompute data. However, materialized views need to be refreshed periodically to ensure the data remains up to date. They are commonly used in data warehousing environments, where quick access to large amounts of pre-aggregated data is essential.

2.4.5 Advantages and Disadvantages of Views

2.4.5.1 Advantages of Views

- ◆ **Simplifies Complex Queries:** Views make it easier to work with complex queries. You can write a complicated SQL query once, and then reuse it in different situations. This means you don't need to repeat the same logic each time you want to retrieve data.
- ◆ **Customizes Data Presentation:** Views allow you to show only the relevant data. For example, if you only need to see certain columns or rows, you can set up a view to display just that, making the data easier to understand and work with.
- ◆ **Enhances Security:** Views can help protect sensitive information by restricting access to only the necessary data. For instance, you can create a

view that shows employee names but hides their salaries, so users only see the information they are allowed to view.

2.4.5.2 Disadvantages of Views

- ◆ **Read-Only in Some Cases:** Some views are read-only, meaning you can not make changes to the data directly through them. If you need to update information, to go back to the original tables, which can be a bit inconvenient.
- ◆ **Performance Issues:** If the view involves joining many tables or large datasets, it may slow down the process. Every time you access a view, the database has to process the query, which can take longer, especially with more complex data.
- ◆ **Data Freshness:** Regular views always show the latest data, but materialized views store data that needs to be manually refreshed. If you rely on materialized views, there might be a delay in seeing the most current information, which could be an issue in time-sensitive environments.

2.4.6 Introduction to Transaction Control Statements

Consider a bank transaction of transferring money from one account to another (say account X to account Y). This consists of following database updates.

- ◆ Debit the amount from account X
- ◆ Update the balance of account X
- ◆ Credit the amount to account Y
- ◆ Update the balance of account Y

The above updates must occur as a logical unit. If all the updates are not performed due to some error or system failure, the integrity of the database will be compromised. To address such problems SQL provides transaction processing features.

2.4.6.1 Definition of Transactions

A transaction is one or more SQL statements that together form a logical unit of work. The statements in transactions are related and perform independent actions. The statements in a transaction are part of a task and all of them are required to complete the task. Transactions have some standard characteristics referred to as ACID properties. The ACID properties of a transaction are:

- ◆ **Atomicity:** Either all operations in a transactions are performed or none of them are performed.
- ◆ **Consistency:** A transaction must transform from one consistent state to another.
- ◆ **Isolation:** Each transaction must execute on its own without interference from other transactions.

- ◆ **Durability:** Once transaction is completed, all changes made by it should be preserved.

2.4.6.2 Importance of Transaction Control in SQL

Transaction control in SQL plays a key role in making sure data stays accurate and consistent when multiple actions are happening at once. It allows you to bundle a series of SQL commands together into one single task. This way, all changes are either completed successfully or not at all. For example, in a bank transfer, if money is being taken from one account and added to another, transaction control makes sure both actions happen together. If something goes wrong, you can use the ROLLBACK command to cancel everything, keeping the data correct. Using transaction control helps to avoid mistakes, especially in busy systems where many users are working with the database at the same time.

2.4.6.3 Basic Transaction States (Started, Committed, Rolled Back)

In SQL, transactions go through three basic states: Started, Committed, and Rolled Back. The transaction begins in the Started state, where changes are made to the database, but they are not saved yet. This is the stage where you can add, modify, or delete data, but nothing is final until you decide what to do next. If everything goes as planned and the changes are correct, the transaction moves to the Committed state. This means all the changes are permanently saved to the database and can be seen by other users. However, if something goes wrong or you need to undo the changes, the transaction can be Rolled Back. In this state, all the changes made during the transaction are canceled, and the database goes back to how it was before the transaction started. These three states help ensure that the database stays accurate and that errors can be fixed easily without affecting the overall system.

2.4.7 Transaction Control Statements

The commands used for transaction control are:

START TRANSACTION: Sets the properties of new transaction and starts the transaction

SET TRANSACTION: Sets the properties of the next transaction to be executed.

SET CONSTRAINTS: sets the constraint mode within a current transaction. The constraint mode controls whether a constraint is applied immediately to data as it is modified or enforcement of the constraint is to be deferred until later in the transaction

SAVEPOINT: Creates a savepoint within a transaction. A savepoint is a place within a transaction's sequence of events that can act as an immediate recovery point. A current transaction can be rolled back to the savepoint instead of the beginning of the transaction

RELEASE SAVEPOINT: Releases a savepoint, freeing up any resources it may be holding

COMMIT: Terminates a successful transaction and commits all changes to a database

ROLLBACK: When used without a savepoint, terminates an unsuccessful transaction and roll back any changes to the beginning of the transaction, restoring the database to its consistent state before the transaction. When used with a savepoint, rollback the transaction to the named savepoint, but allows it to continue

2.4.8 Advantages of Transaction Control

- ◆ **Maintains Data Integrity:** Transaction control helps keep your data accurate and consistent. If something goes wrong during a transaction, you can roll back the changes to make sure no incorrect data is saved in the database.
- ◆ **Ensures Atomicity:** With transaction control, all actions in a transaction are treated as one unit. This means that either all the changes will happen, or none of them will, ensuring your database remains in a valid state even if an error occurs.
- ◆ **Guarantees Consistency:** It ensures that the database always moves from one valid state to another. If an error happens, the transaction can be rolled back, preventing partial updates from causing problems.
- ◆ **Provides Isolation:** When one transaction is running, others can not interfere with it. This helps avoid data conflicts or inconsistencies when multiple users are working with the database at the same time.
- ◆ **Ensures Durability:** Once a transaction is committed, its changes are permanent. Even if the system crashes or there is a power failure, the changes made in the transaction are safe and won't be lost.
- ◆ **Handles Errors Easily:** If an error happens during a transaction, transaction control lets you undo the changes and revert to the original state. This makes managing errors easier and helps protect the database from issues.
- ◆ **Supports Multiple Users:** Transaction control makes it easier for many people to use the database at the same time without causing problems. Each transaction runs independently, so multiple users can work without affecting each other's data.
- ◆ **Improves Data Management:** It helps organize large sets of changes by grouping them into a single transaction. If an error happens, only that transaction is affected, not others, making it simpler to manage changes to the data.

2.4.9 Disadvantages of Transaction Control

- ◆ **Complexity in Management:** Using transaction control can make managing the database more complicated. In systems with many users and transactions, it can be tricky to keep track of when transactions start, when they are committed, or when they should be rolled back. This extra complexity can lead to mistakes.
- ◆ **Performance Overhead:** Transaction control requires extra processing to make sure changes are either saved or undone. This extra work can slow down the system, especially when handling large or complicated transactions, affecting the overall performance of the database.

- ◆ **Potential for Data Inconsistency:** If transactions are rolled back at the wrong time or in the wrong way, it can cause inconsistencies in the data. This is particularly risky in systems where multiple transactions happen at once, as it can confuse the system and cause errors in the data.
- ◆ **Resource Consumption:** To manage transactions, the system often needs to keep logs and other records to ensure it can undo changes if needed. This can use up a lot of system resources, like memory and storage, which can become an issue when there are a lot of transactions happening frequently.
- ◆ **Difficulty in Handling Deadlocks:** In busy systems with many transactions happening at the same time, deadlocks can occur, where two or more transactions block each other. Resolving these deadlocks and making sure the system keeps running smoothly can be difficult and requires extra resources.

2.4.10 Practical Applications of Views and Transaction Control

- ◆ **Simplifying Complex Queries:** Views are great for making complicated queries easier to use. Instead of writing the same complex query every time, you can save it as a view and reuse it whenever needed. This saves time and effort.
- ◆ **Data Security:** Views help keep sensitive information safe by showing only the data that is needed. For example, if you have employee data, you could create a view that shows only their names and roles, but hides their salary or personal information from unauthorized users.
- ◆ **Data Presentation:** Views allow you to organize data in a way that makes sense for the situation. For instance, you could create a view that shows sales figures by region or department, without overwhelming users with unnecessary details like individual transactions.
- ◆ **Enhancing Data Consistency:** Views help ensure that everyone is looking at the same data in the same way. They simplify how data is accessed, reducing errors that could happen if people wrote their own complex queries.
- ◆ **Transaction Control for Data Integrity:** Transaction control ensures that changes to the database are safe and reliable. With commands like COMMIT, ROLLBACK, and SAVEPOINT, you can make sure that if something goes wrong, changes can be undone and the database remains accurate.
- ◆ **Error Recovery:** If there is an error during a transaction, you can use ROLLBACK to undo the changes and restore the database to its previous state. This helps prevent mistakes from affecting the system.
- ◆ **Ensuring Atomicity in Operations:** Transactions make sure that a group of changes happens together or not at all. If one part of the operation fails, the whole transaction is canceled, ensuring that the database stays in a good state.
- ◆ **Multiple Users and Concurrency:** When many people are working with

the database at the same time, views and transaction control help ensure that their actions do not interfere with each other. This keeps everything running smoothly and prevents data conflicts.

- ◆ **Data Reporting:** Views are helpful for creating reports. They can gather and display data from different parts of the database in a way that is easy to read and understand, without changing the original data.
- ◆ **Transactional Workflow Management:** In systems like banking or inventory management, transaction control makes sure that important actions—like transferring money or updating stock levels—are done correctly. If something goes wrong, it ensures no partial changes are made, keeping everything consistent.

Recap

- ◆ A view is a virtual table that displays specific data from one or more tables without storing it.
- ◆ Views help simplify complex queries, improve security, and provide data abstraction.
- ◆ Unlike tables, views do not store data but provide a read-only representation unless explicitly made updatable.
- ◆ Views are created using the CREATE VIEW statement, e.g., CREATE VIEW FACULTY AS SELECT ID, NAME FROM INSTRUCTOR;.
- ◆ Views can be updated only if they follow certain conditions, such as using a single table and not containing aggregations.
- ◆ Types of Views: Simple View (single table), Complex View (multiple tables), and Materialized View (stores results physically).
- ◆ Views enhance security by restricting access to sensitive data but can also slow performance in some cases.
- ◆ A transaction is a sequence of SQL operations that execute as a single unit of work.
- ◆ Transactions follow the ACID properties: Atomicity (all or nothing), Consistency (valid state transition), Isolation (independent execution), and Durability (permanent changes).
- ◆ The main transaction states are: Active (processing), Committed (saved), and Rolled Back (undone in case of failure).
- ◆ The COMMIT command permanently saves changes made during a transaction.

- ◆ The ROLLBACK command undoes changes made since the last COMMIT.
- ◆ SAVEPOINT allows setting intermediate rollback points within a transaction for partial rollbacks.
- ◆ Transaction control ensures data integrity and prevents errors when multiple users interact with the database simultaneously.
- ◆ Practical applications: Used in banking (ensuring successful fund transfers), inventory management, and financial reporting to maintain data consistency.

Objective Type Questions

1. What SQL command is used to create a view?
2. What keyword is used to remove a view from the database?
3. What SQL clause is used to define the columns retrieved in a view?
4. What type of view stores the result physically?
5. Which SQL command is used to undo a transaction?
6. What keyword is used to permanently save a transaction?
7. What happens to uncommitted transactions when a database crashes?
8. What command is used to set a temporary transaction point?
9. What does ACID stand for in database transactions?
10. Which ACID property ensures that a transaction is either fully completed or fully rolled back?

Answers to Objective Type Questions

1. CREATE VIEW
2. DROP VIEW
3. SELECT
4. Materialized View
5. ROLLBACK
6. COMMIT

7. They Are Rolled Back
8. SAVEPOINT
9. Atomicity, Consistency, Isolation, Durability
10. Atomicity

Assignments

1. Explain how views can be used to simplify complex database queries. Discuss their role in improving data retrieval and user experience, and provide examples where views can enhance the efficiency of large-scale data systems.
2. Analyze the advantages and disadvantages of using views in SQL. How can views help with data security and access control? In what scenarios could the limitations of views affect their usability in a real-world application?
3. Discuss the role of transaction control statements like COMMIT, ROLLBACK, and SAVEPOINT in maintaining data integrity. How do these statements ensure that transactions are executed properly, especially in high-concurrency environments? Provide examples from database-driven applications where these controls are crucial.
4. Evaluate the impact of using materialized views in SQL for improving performance. Compare materialized views with regular views in terms of data refresh cycles, storage requirements, and application performance. When would materialized views be preferable?
5. Consider a scenario where multiple users are interacting with a database simultaneously. Explain how transaction control mechanisms like atomicity, consistency, isolation, and durability (ACID properties) ensure that the database remains in a consistent state. How do these properties apply to SQL views in multi-user environments?

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems*. McGraw-Hill.
2. Coronel, C., & Morris, S. (2019). *Database systems: design, implementation, and management*. Cengage Learning.
3. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2006). *Distributed databases*. In *Database system concepts* (5th ed., pp. 705–749). McGraw-Hill.
4. Groff, J. R., Weinberg, P. N., & Oppel, A. J. (2002). *SQL: The complete reference* (2nd ed.). McGraw-Hill/Osborne.

Reference

1. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage Learning.
2. Pratt, P. J., & Last, M. Z. (2020). *Concepts of database management* (10th ed.). Cengage Learning.
3. Oppel, A. J. (2015). *SQL: A beginner's guide* (4th ed.). McGraw-Hill Education
4. Libeskind, R. (2020). *Understanding SQL: A beginner's guide* (1st ed.). Independently published.
5. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill Education.



PL/SQL



Unit 1

Introduction to PL/SQL

Learning Outcomes

Studying this unit will enable the student to:

- ◆ define PL/SQL block structure
- ◆ define variables, conditions, and loops in PL/SQL
- ◆ describe Handle errors with PL/SQL exceptions
- ◆ detail the PL/SQL code block
- ◆ describe Control program flow with IF statements and loops

Prerequisites

We study PL/SQL because it enhances the capabilities of SQL by adding procedural programming features such as variables, loops, and conditional statements, allowing for more complex and efficient database operations. For example, in SQL, you can retrieve data from a table, but with PL/SQL, you can use variables and loops to process that data, make decisions based on conditions, and handle errors within the same block of code. For instance, if you wanted to calculate the total salary for employees in a specific department and display a message based on whether the total exceeds a certain threshold, you could easily achieve this using PL/SQL. Understanding PL/SQL allows developers to write more efficient, maintainable, and error-resilient code for database applications, especially when working with Oracle databases.

Keywords

PL/SQL, Oracle, SQL, Datatype, Variable, Conditional Statement, LOOP



Discussion

Structured Query Language, popularly known as SQL (pronounced as SEQUEL), is a query language used to create and manipulate relational databases. In practical applications database access is implemented through software programs. These software programs are created using general purpose programming languages such as, C, C++, JAVA etc. SQL commands for database access are embedded in these programs. SQL is the natural language of relational database management systems.

3.1.1 Disadvantages of SQL

Although SQL is widely used, it does not support programming features like condition checking, looping, and branching. These features are important for verifying data integrity before storing it in a database.

In multi-user environments, SQL statements can slow down data processing. Each time an SQL query runs, the database engine is called, which can increase network traffic. SQL also lacks built-in error-handling mechanisms. If an error occurs, the database engine displays its own predefined error messages, which cannot be customized by the user.

All these disadvantages prevent SQL from being a fully structured programming language. Oracle, most popular relational DBMS, provides a fully structured programming language called PL/SQL.

3.1.2 Advantages of PL/SQL

- ◆ Provides the procedural capabilities like condition checking, branching and looping.
- ◆ The entire block of code is passed to the Oracle engine in one go. Hence the network traffic is reduced considerably.
- ◆ PL/SQL provides facilities for displaying user-friendly error messages.
- ◆ PL/SQL allows the use of variables to store results of queries.
- ◆ Applications written in PL/SQL can be executed in any platform, where Oracle is operational.

3.1.3 PL/SQL Block

A PL/SQL block consists of SQL statements organized in a structured manner. A **PL/SQL block** is the fundamental structure of a PL/SQL program, consisting of four main sections:

- ◆ The Declare section

- ◆ The Begin Section
- ◆ The Exception Section
- ◆ The End Section

The following fig 3.1.1 shows PL/SQL block structure

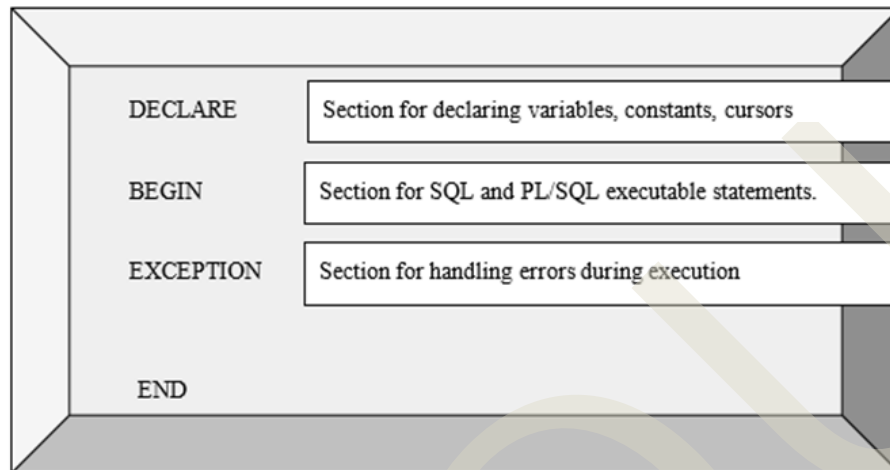


Fig. 3.1.1 PL/SQL block structure

PL/SQL engine resides in Oracle engine. Oracle engine can process the entire PL/SQL blocks. The fig 3.1.2 shows the PL/SQL execution environment.

A PL/SQL block starts with the keyword **DECLARE** (if declarations are included) and ends with **END;**. This structure ensures modular, readable, and maintainable code.

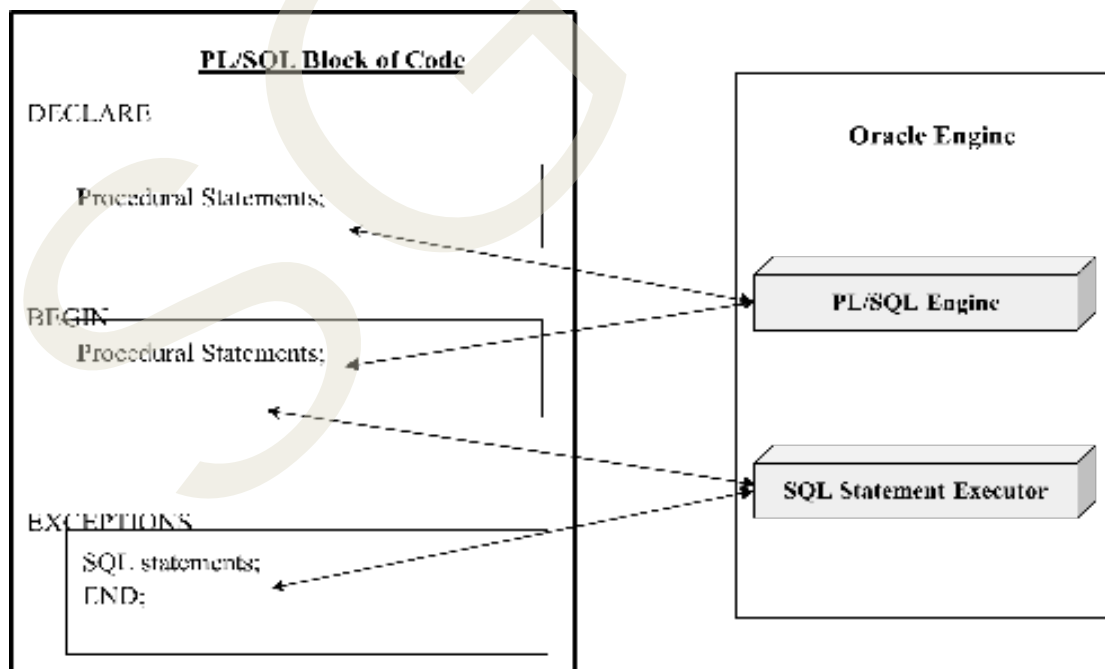


Fig. 3.1.2 PL/SQL block of code

3.1.4 PL/SQL Character Set

The **PL/SQL character set** includes a collection of symbols, letters, and special characters that can be used in PL/SQL programs. It consists of:

1. **Letters:** Uppercase (A-Z) and lowercase (a-z) alphabets.
2. **Digits:** Numbers from 0 to 9.
3. **Symbols:** Special symbols like +, -, *, /, =, >, <, and more.
4. **Whitespace Characters:** Spaces, tabs, and newline characters for readability.

This character set ensures compatibility with SQL and facilitates smooth interaction with database objects.

3.1.5 PL/SQL Literals and Constants

In PL/SQL, **literals** are fixed values directly written in the code, such as numbers, strings, or dates, which do not change during execution. For example, **42**, **'Hello'**, and **SYSDATE** are literals. **Constants**, on the other hand, are named variables whose value is assigned once and cannot be modified later in the program. They are declared using the keyword **CONSTANT** and help ensure data integrity by preventing unintended changes. Both literals and constants are fundamental for defining fixed values in PL/SQL programs.

- ◆ Literal is a character string or numeric used to represent itself.
- ◆ The following table shows literals supported by PL/SQL

Table 3.1.1 Literals supported by PL/SQL

Literal Type	Example
Numeric - Integers or floats	37, 1.2, 1.e5, 8g3
String-one or more characters enclosed in single quotes	'Hello', 'C', '*',
Boolean-Predetermined Constants	TRUE, FALSE, NULL
Date and Time literals	DATE'2022-01-01'; TIMESTAMP'2022-01-01 10:00:01';

3.1.6 PL/SQL Data Types

PL/SQL provides various **data types** to define variables, constants, and parameters, ensuring compatibility with database operations. These data types are categorized as follows:

1. **Scalar Data Types:** Represent single values such as numbers (**NUMBER**, **INTEGER**), characters (**CHAR**, **VARCHAR2**), and dates (**DATE**).
2. **Composite Data Types:** Store multiple values in collections or records, like

TABLE, VARRAY, and RECORD.

3. LOB Data Types: Handle large objects such as **CLOB** (Character Large Object) and **BLOB** (Binary Large Object).

4. Reference Data Types: Include pointers like **REF CURSOR**.

These data types provide flexibility for handling diverse data in PL/SQL programs. See the table

Table 3.1.2 PL/SQL Qata Types

Default Data Types	Number, Char, Date, Boolean
%TYPE	Declare variables based on definitions of columns in the database tables.
%ROWTYPE	Composite Data Type. Similar to structure in C language. Declare a record variable based on the database table.
BLOB-Binary Large Objects	Stores unstructured binary data upto 4GB.
PLS_INTEGER	Used for whole numbers in arithmetic operations. Not stored in the database.
Number(p,s)	Used for storing floatingpoint values 'p' represents the total number of digits allowed for the value 's' number of digits right to the decimal place.

3.1.7 DBMS_OUTPUT Package

The DBMS_OUTPUT package is used in PL/SQL to display messages from a program. It helps in debugging and understanding the flow of execution by showing output during or after program execution.

Messages written using DBMS_OUTPUT are temporarily stored in a buffer. Once the program finishes running, these messages are retrieved and displayed on the screen.

The package includes several procedures to manage and display output, enabling developers to effectively track and debug their PL/SQL code.

Procedures in DBMS_OUTPUT package are given below

Table 3.1.3 procedures in DBMS_OUTPUT package

Procedure	Purpose
DBMS_OUTPUT.ENABLE	Allow messages to display
DBMS_OUTPUT.DISABLE	Does not allow messages to display
DBMS_OUTPUT.PUT	Places information in the buffer
DBMS_OUTPUT.PUT_LINE	Places information in the buffer with an end-of-line marker
DBMS_OUTPUT.NEW_LINE	Places an end-of -line marker in the buffer

Example of DBMS - OUTPUT Package

PL/SQL Block	OUTPUT
<pre>BEGIN DBMS_OUTPUT.PUT('This is'); DBMS_OUTPUT.PUT_LINE (This is line 1.); DBMS_OUTPUT.PUT('This is line 2. '); END;</pre>	<pre>This is . This is line 1. This is line 2.</pre>

PL/SQL Block	OUTPUT
<pre>DECLARE PI CONSTANT NUMBER:=3.14; radius NUMBER(5, 2); area NUMBER(10, 2); BEGIN radius:=9.5; area:=PI*radius*radius; DBMS_OUTPUT.PUT_LINE('The area of the circle is ' area); END; /</pre>	<pre>The area is 283.53 PL/SQL procedure successfully executed</pre>

3.1.8 PL/SQL Basic Syntax

Example: Write a PL/SQL block to print HELLO WORLD

PL/SQL Block	OUTPUT
<pre>BEGIN DBMS_OUTPUT.PUT_ LINE('HELLO WORLD') END;</pre>	<pre>HELLO WORLD PL/SQL procedure successfully</pre>

- ◆ 'END;' marks the end of the block.

Activity 1: Write a PL/SQL block to print your name

Comments in PL/SQL

- ◆ Comments are used to improve the readability of code.
- ◆ In PL/SQL single line comments comments with '--';
- ◆ Block comments or multi line comments can be included using '/*...*/'.

3.1.9 Variable Declaration in PL/SQL

Syntax:

variable_name [CONSTANT] datatype [NOT NULL]:=value;

where,

- ◆ variable_name : The name of the variable
- ◆ [CONSTANT]: Optional. If a variable is declared as CONSTANT the value cannot be changed.
- ◆ datatype- can be the default data type or %TYPE or %ROWTYPE.
- ◆ [NOT NULL]: Optional. Specified when the variable cannot have null values.
- ◆ value:value assigned to the variable'

Examples:

```
radius NUMBER(2);
Area NUMBER(5,2);
Pi CONSTANT NUMBER:=3.14;
Grade CHAR(1) DEFAULT 'A';
Marks NUMBER NOT NULL;
```

The above examples show various types of variable declarations. Default values can be assigned as shown in example 4.

- ◆ Every PL/SQL statement ends with a semicolon;
- ◆ ':= ' is used as assignment operator;

Example: Write a PL/SQL block that finds the area and circumference of a circle.

The above example computes the area of a circle given the radius 9.5.

In PL/SQL input can be accepted from the user directly. Consider the following example.

PL/SQL Block	OUTPUT
<pre>DECLARE PI CONSTANT NUMBER:=3.14; radius NUMBER; area NUMBER; BEGIN radius:=&radius; area:=PI*radius*radius;</pre>	<pre>The area is of the circle is 314 PL/SQL procedure successfully executed</pre>

```
DBMS_OUTPUT.PUT_LINE('The area  
of the circle is '||area);  
END;
```

In the DECLARE section the variable radius is declared. In the BEGIN section, the value for variable is accepted from the user.

3.1.10 PL/SQL Operators

The following table shows different types of operators in PL/SQL

Table 3.1.4 Arithmetic Operators

Arithmetic Operations	Description
+	Addition (a+b)
-	Subtraction (a-b)
*	Multiplication (a*b)
/	Division (a/b)
**	Exponentiation (a ^b)

Table 3.1.5 Relational Operators

Relational Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!= or <>	Not equal to
=	Equal to

Table 3.1.6 Special Operators

Operators	Description	Example
LIKE	Pattern Matching Compare a character or string to a pattern specified. Return TRUE if the value matches the pattern and FALSE if it does not	'Suma' like 'S%a' returns a Boolean true, because the string starts with S and ends with A. Otherwise it will return FALSE, '%' is used to match any number of characters.
BETWEEN	To specify a range	SALARY BETWEEN 10000 AND 50000

IN	To test whether an item is present in a list of values	If a='x' then, a in ('r', 's', 't') returns Boolean false but a in ('x', 'y', 'z') returns Boolean true.
IS NULL	To test whether the variable has Null value. Returns TRUE, if the value of a variable is NULL. Otherwise returns FALSE.	a is null

Table 3.1.7 Logical Operators

Logical Operator	Description
AND	Logical AND Returns TRUE only if both conditions are TRUE
OR	LOGICAL OR Returns TRUE if any of the conditions are evaluates to TRUE
NOT	Negation. Used to reverse the logical state of the operand

- ‘||’ operator is used as concatenation operator.

3.1.11 PL/SQL Conditional Statement

Conditional statements in PL/SQL are used to make decisions based on specified conditions. Like other programming languages, PL/SQL uses the IF statement to evaluate conditions. These conditions are represented as boolean expressions, which return either TRUE or FALSE.

PL/SQL supports the following conditional constructs for decision-making:

- ◆ IF-THEN
- ◆ IF-THEN-ELSE
- ◆ IF-THEN-ELSIF-ELSE

Syntax and Explanation

IF-THEN: Executes a block of code if the condition evaluates to **TRUE**.

Syntax and Example:

Syntax

```
IF condition THEN
    Statements;
END IF;
```

Sample Code

```
IF(a>b) THEN
    larger:=a;
END IF;
```

5. If the condition evaluates to TRUE, the statements within the corresponding IF block will be executed.
 6. If the condition evaluates to FALSE, the statements inside the IF block will be skipped. If there is an ELSE or ELSIF block, the program will evaluate those accordingly.
- ◆ **IF-THEN-ELSE:** Provides an alternative block of code to execute when the condition is FALSE. Here ELSE provides an alternative. If the condition specified in IF is FALSE, the ELSE part will be executed.

Syntax

```
IF condition THEN
-- statements if condition is true
ELSE
-- statements if condition is false
END IF;
```

IF-THEN-ELSIF-ELSE: Allows multiple conditions to be checked sequentially.

```
IF condition1 THEN
-- statements for condition1
ELSIF condition2 THEN
-- statements for condition2
ELSE
-- statements if none of the above conditions are true
END IF;
```

Example

```
DECLARE
grade NUMBER := 85;
BEGIN
IF grade >= 90 THEN
DBMS_OUTPUT.PUT_LINE('Grade: A');
ELSIF grade >= 75 THEN
DBMS_OUTPUT.PUT_LINE('Grade: B');
ELSE
```

```

DBMS_OUTPUT.PUT_LINE('Grade: C');

END IF;

END;

```

This PL/SQL block evaluates the grade variable and outputs the appropriate grade message based on the value.

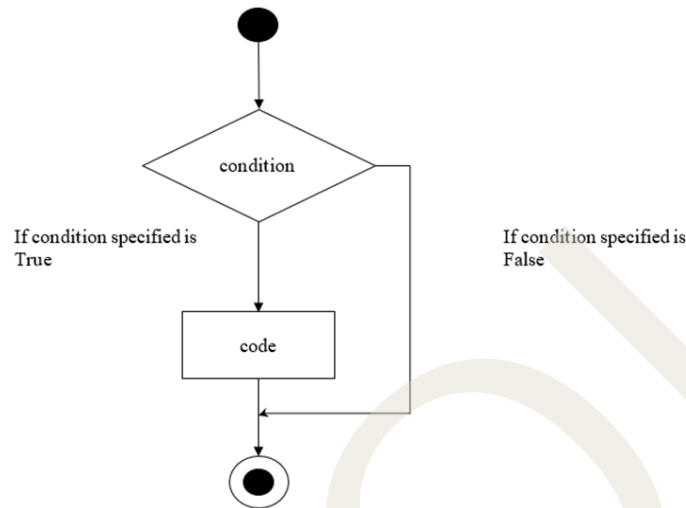


Fig. 3.1.3 Flow chart of Conditional Statement

The above flowchart represents the general form of conditions constructed in all programming languages. The general syntax for IF statement in PL/SQL is :

Syntax

```

IF<condition> THEN
    <PL/SQL statements>
ELSIF<condition> THEN
    <PL/SQL statements>
ELSE
    <PL/SQL statements>
END IF;

```

IF-THEN-ELSIF statement

- ◆ Provides the facility to choose between several alternatives.
- ◆ IF-THEN statement followed by an optional ELSIF-ELSE statement.

Syntax	Sample Code
IF condition1 THEN	DECLARE color varchar2(6):='Red';

Statements;	BEGIN
ELSIF condition2	IF (color:='Green') THEN
THEN	DBMS_OUTPUT.PUT_LINE:('Your car is
	Green');
Statements;	ELSIF:(color:='White') THEN
ELSIF condition3	DBMS_OUTPUT.PUT_LINE:('Your car is
	White');
THEN	ELSIF:(color:='Red') THEN
Statements;	DBMS_OUTPUT.PUT_LINE:('Your car is
	Red');
	ELSE
ELSE	DBMS_OUTPUT.PUT_LINE:('Your car is
	not Red or Green or White');
Statements;	DBMS_OUTPUT.PUT_LINE:('Your car is in
	' color 'color');
END IF;	END IF;
	END;
	/

- ◆ IF statements can be nested. An IF statement can be written inside another IF statement.

3.1.12 Loop Statements

Loops are used to execute a set of statements multiple times and are an essential control structure in all programming languages. They help automate repetitive tasks, reduce code redundancy, and improve program efficiency. A loop generally involves four key steps: initialization, where the starting point is set; a condition that determines whether the loop continues; execution of the loop body if the condition evaluates to TRUE; and an update that modifies variables to ensure the condition is eventually met. This structure ensures controlled and efficient repetition.

- In PL/SQL there are mainly three types of LOOPS:
 - Simple LOOP
 - FOR LOOP
 - WHILE LOOP

3.1.12.1 Simple LOOP

- ◆ A sequence of statements are enclosed between LOOP and END LOOP statements.

◆ Syntax and Example

Syntax	Sample Code
LOOP	DECLARE
-----Sequence of Statements	i number:=0;
EXIT WHEN condition;-----	BEGIN
----condition to terminate the	LOOP
LOOP	i:=i+2;
END LOOP	EXIT WHEN i>10;
	END LOOP
	<u>Output</u>
	The value of i is 10

In the above program variable i is initialized as 0. In each iteration the value of i is incremented by 2. The loop will be terminated as soon as the value i is greater than 10. The statement after the loop will display the final value of i ie.,10.

3.1.12.2 FOR LOOP

- ◆ Executes the statements inside the loop for a specific number of times.

Syntax	Sample Code
FOR Variable IN	DECLARE
start.... end	i number:=0;
LOOP	BEGIN
Statements;	FOR i IN 0..5
END LOOP;	LOOP
	DBMS_OUTPUT.PUT_LINE(i);
	END LOOP;
	END;

Output

0
1
2
3
4
5

PL/SQL procedure completed successfully

The above program displays the value of i in each iteration. Initially the value of i is 0. So 0 will be displayed. In the next iteration the value of i is incremented by 1. So 1 will be displayed. Likewise the values of i will be displayed till i reaches 5. After that the loop will be terminated.

- The variable in the FOR LOOP need not be declared.
- The loop variable is incremented by 1.

3.1.12.3 FOR LOOP REVERSE

Syntax	Sample Code
FOR Variable IN REVERSE	DECLARE
start---end	i number(2);
LOOP	BEGIN
Statements;	FOR i IN REVERSE 1.....10
END LOOP;	LOOP
	DBMS_OUTPUT.PUT_LINE(i);
	END LOOP;
	END;
	<u>Output</u>
	10
	9
	8
	7
	6
	5
	4
	3
	2
	1

PL/SQL procedure completed successfully

In PL/SQL, the **REVERSE** keyword is used to reverse the order of iteration in a loop. When using the FOR loop with the **REVERSE** keyword, the loop variable starts from the highest value and decrements with each iteration until it reaches the lowest value.

The **REVERSE** keyword causes the loop to iterate in descending order, starting from 10 and ending at 1. During each iteration, the loop variable i is decremented automatically. The above program displays the value of i in each iteration in the reverse order.

3.1.12.4 WHILE LOOP

- ◆ This type of loop is commonly known as a WHILE loop in PL/SQL and other programming languages. It is used in situations where the number of iterations is not known beforehand. The loop continues to execute as long as the specified condition evaluates to TRUE. Unlike a FOR loop, the loop variable in a WHILE loop needs to be explicitly incremented or decremented within the loop body to ensure proper control of the loop.

Syntax and Example

Syntax
WHILE condition
LOOP
Statements;
ENDLOOP

Sample Code
DECLARE
 i number:=0;
BEGIN
 WHILE i<5
 LOOP
 DBMS_OUTPUT.PUT_LINE(i);
 i=i+1;
 END LOOP;
END;

Output

0
1
2
3
4
5

PL/SQL procedure completed successfully

In the above example i is initialized to 0. When the iteration starts, the value of i is checked. i<5 is TRUE. The value of i is displayed and i is incremented by 1. The new value of i is 1. Again the condition is checked and i<5 holds TRUE. The value of i is displayed as 1. i is incremented by 1. The new value of i is 2. The loop continues as long as i<5 holds TRUE. Once the value i is greater than 5, the loop will be terminated.

Anchored Declaration: %TYPE

- ◆ Used to declare variables based on definitions of columns in the database tables.
- ◆ Usage 1: Declare variables that directly map to column definitions in the database. Hence datatype need not be specified explicitly. Constraints will not be copied.
 - Syntax: variable_name TABLENAME COLUMN NAME
% TYPE;

- Example: `ename EMPLOYEE.NAME%TYPE;`

Here EMPLOYEE is a database table, NAME is a column in EMPLOYEE. The data type of NAME is applied to ename. Any change in the data type or precision for the NAME column in the EMPLOYEE table will be applied automatically to the variable anchored.

- ◆ Usage 2: To declare a variable whose data type that directly maps to a datatype of a previously declared variable.

The data type and NOT NULL constraint of name is applied to the variable ename. If the value is assigned to the name variable, it will not be copied.

Composite Data type %ROWTYPE

- ◆ Used to store multiple column values in a single name.
- ◆ Similar to the concept of structure in C language.
- ◆ Allows to declare a record variable based on the column definitions in the database table.
 - Syntax: `recordvariablename tablename%ROWTYPE;`
 - Example: `emprec EMPLOYEE%ROWTYPE;`

The above declaration creates a record variable for the EMPLOYEE table in the database. Each column in the table can be accessed using the record variable.

Using SELECT statement in PL/SQL

- ◆ Values from database tables can be fetched to PL/SQL variables using the SELECT INTO statement.
- ◆ Only one row value can be returned to the variable.
- ◆ If no row is returned then the 'No Data Found' exception is thrown.
- ◆ If more than one record is returned, then the exception 'Exact fetch returns more than requested number of records' is thrown.

Example: An EMPLOYEE table is given below.

Table 3.1.8 Employee Table

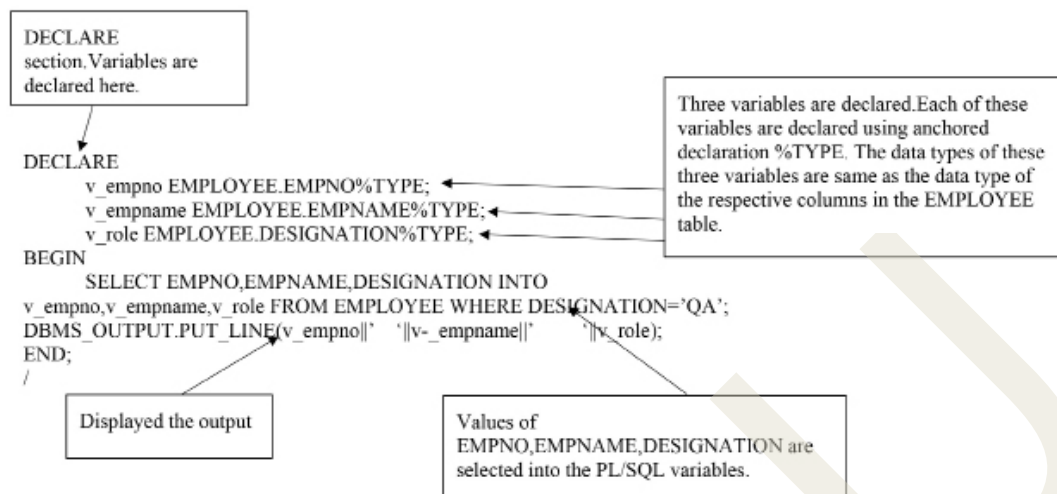
EMP NO	EMP NAME	DESIGNATION	STATE	SALARY	DEPT NO
E0001	JACOB	SYSADMIN	MAHARASHTRA	16500	D0001
E0002	JOHN	MANAGER	TAMILNADU	32000	D0002
E0003	ABRAHAM	DEVELOPER	KARNATAKA	26500	D0002
E0004	JOY	QA	TAMILNADU	19560	D0001
E0005	CHRISTY	SECRETARY	MAHARASHTRA	13200	D0003

Our aim is to select the name and employee number QA

The actual SQL query will be:

SELECT EMPNO, EMPNAME, DESIGNATION FROM EMPLOYEE WHERE DESIGNATION="QA";

Let us display the details of the particular employee using a PL/SQL block.



OUTPUT

E0004 JOY QA
PL/SQL procedure successfully completed.

Recap

- ◆ SQL is used for managing relational databases but lacks procedural features like looping, condition checking, and error handling.
- ◆ PL/SQL extends SQL by adding procedural capabilities (e.g., variables, loops, conditionals).
- ◆ PL/SQL reduces network traffic by sending entire code blocks at once to the Oracle engine.
- ◆ It provides error handling and user-friendly error messages through the `EXCEPTION` section.
- ◆ The `DBMS_OUTPUT` package allows output for debugging.
- ◆ PL/SQL blocks consist of `DECLARE`, `BEGIN`, `EXCEPTION`, and `END` sections.
- ◆ Variables can be declared using anchored types like `%TYPE` and `%ROWTYPE` for easy mapping with database columns.

Objective Type Questions

1. What does SQL stand for?
2. Which programming language extension is used for procedural capabilities in Oracle?
3. What section of a PL/SQL block handles errors?
4. What keyword is used to declare constants in PL/SQL?
5. Which function is used in PL/SQL to display output?
6. What is the primary data type for numbers in PL/SQL?
7. What symbol is used for string concatenation in PL/SQL?
8. What type of loop executes statements a fixed number of times?
9. What keyword is used to exit a loop in PL/SQL?
10. Which section of a PL/SQL block holds the executable statements?

Answers to Objective Type Questions

1. STRUCTURED
2. PL/SQL
3. EXCEPTION
4. CONSTANT
5. DBMS_OUTPUT
6. NUMBER
7. ||
8. FOR
9. EXIT
10. BEGIN

Assignments

1. Write a program PL/SQL block that adds two numbers and displays the result.
2. Write a PL/SQL program that finds the largest among three numbers.
3. Write a PL/SQL program to find the cube of a number.
4. Write a PL/SQL program to print the multiplication table of a number.
5. Write a PL/SQL program which include "for loop"

Suggested Reading

1. Bayross, I. (n.d.). *SQL, PL/SQL: The programming language of Oracle* (3rd ed.). BPB Publications.
2. Feuerstein, S. (n.d.). *Oracle PL/SQL programming*. O'Reilly Media.
3. McDonald, C., Katz, C., Beck, C., Kallman, J. R., & Knox, D. C. (n.d.). *Mastering Oracle PL/SQL: Practical solutions*. Apress Media.
4. Oracle Tutorial. (n.d.). *PL/SQL tutorial*. Retrieved March 5, 2025, from <https://www.oracletutorial.com/plsql-tutorial/>

Reference

1. [geeksforgeeks.org/pl-sql-tutorial/](https://www.geeksforgeeks.org/pl-sql-tutorial/)
2. archive.nptel.ac.in/courses/106/105/106105175/
3. tutoxialspoint.com/plsql/index.htm

Unit 2

Cursors

Learning Outcomes

The learner will be able to:

- ◆ define the concept of a cursor in PL/SQL and its role in managing multiple rows of data
- ◆ identify the two types of cursors: implicit and explicit
- ◆ describe how a cursor loop works
- ◆ describe the automatic management of implicit cursors
- ◆ describe the process of declaring, opening, fetching, and closing explicit cursors

Prerequisites

To effectively work with PL/SQL, it is essential to understand the concepts of variables, loops, and conditional statements, which allow you to perform dynamic operations on data. For example, consider a restaurant that needs to apply discounts to customers' bills based on their total spending. The rules are simple: if a customer spends more than \$100, they receive a 20% discount; if they spend between \$50 and \$100, they get a 10% discount; and if their bill is below \$50, no discount is applied. Using PL/SQL, this process can be handled row by row, just like a waiter serving dishes individually to each customer. In this scenario, a cursor would fetch each customer's bill, a loop would iterate over the data, and conditional statements would apply the correct discount. Finally, the updated bill amount would be stored back in the database, ensuring each customer receives the correct discount based on their individual order. This is similar to how variables, loops, and conditions in PL/SQL work together to efficiently manage data and perform operations on a row-by-row basis, which SQL alone cannot accomplish efficiently.

Keywords

Cursor, Active dataset, Implicit Cursor, Explicit Cursor.



Discussion

In PL/SQL, a cursor is a database object used to retrieve and process multiple rows of data from a query, one row at a time. It acts as a pointer that allows the program to navigate through the result set of a query, providing control over how data is fetched and processed. Cursors are especially useful when you need to handle complex queries or perform operations on individual rows. There are two main types of cursors in PL/SQL: implicit cursors, which are automatically created by the system for single SQL queries, and explicit cursors, which are manually declared and controlled by the programmer for more complex operations.

SQL statements commands operate the rows in the result in one go . Consider the following table EMPLOYEE.

Table 3.2.1 Employee Table

EMP NO	EMP NAME	DESIGNATION	STATE	SALARY	DEPT NO
E0001	JACOB	SYSADMIN	MAHARASHTRA	16500	D0001
E0002	JOHN	MANAGER	TAMILNADU	32000	D0002
E0003	ABRAHAM	DEVELOPER	KARNATAKA	26500	D0002
E0004	JOY	QA	TAMILNADU	19560	D0001
E0005	CHRISTY	SECRETARY	MAHARASHTRA	13200	D0003

If we want to update the salary of each employee by 10%, the following SQL query can be executed. .

```
UPDATE EMPLOYEE SET SALARY=SALARY+(SALARY*(10/100));
```

The above query updates all the values in the SALARY column with an increment of 10%. What if the percentage of increment is different for different values of salary? ie, Update SALARY by 10% whose current salary is between 10000 and 20000. Similarly, update SALARY by 15 whose current salary is between 21000 and 30000. In order to update salaries with different values, multiple SQL queries have to be executed. The rows are to be updated one by one by checking the condition. In such situations we can create a work area in Oracle, store the result set and update the rows one by one.

3.2.1 Cursor

For every query executed, Oracle creates a work space, known as the context area, for processing an SQL statement. The context area contains all the information needed for processing the SQL statement. For example, the number of rows processed. In PL/SQL this context area is called Cursor

A Cursor is a memory area used by the Oracle engine for its internal processing in order to execute SQL statements.

- ◆ Cursor points to the context area.

- ◆ The context area contains the following information:
 - rows returned by the SQL query.
 - number of rows processed by the SQL query.
 - A pointer to the parsed query.
- ◆ The data stored in the cursor is called Active Dataset.

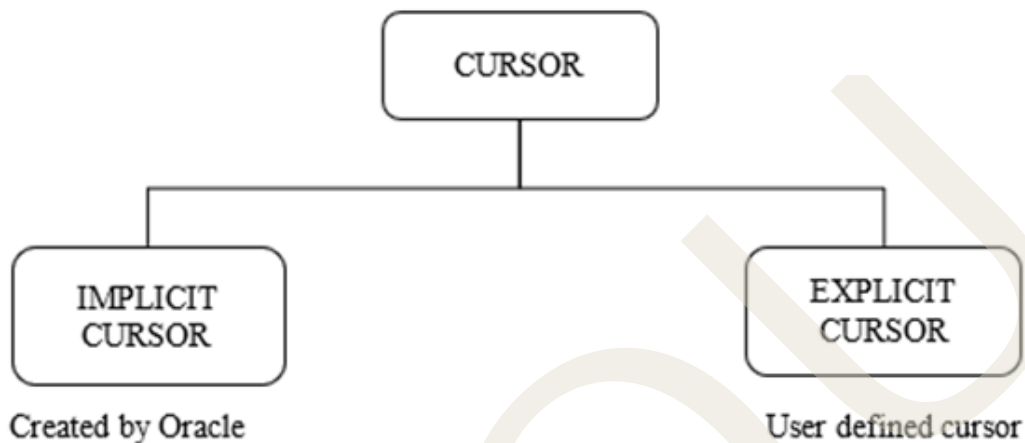


Fig. 3.2.1 Types of cursors

3.2.2 Implicit Cursor

An **implicit cursor** is automatically created and managed by the Oracle engine when executing SQL statements like `SELECT INTO`, `INSERT`, `DELETE`, or `UPDATE`. It is used to handle the execution status of these SQL statements, such as whether the operation was successful, how many rows were affected, or if there were any errors.

The entire execution cycle of implicit cursors—opening, fetching, and closing—is managed by the Oracle engine, so the programmer doesn't need to explicitly define or control it. Implicit cursors are ideal for simple SQL operations where complex row-by-row processing isn't required.

The following table shows Implicit Cursor attributes and their meaning

Table 3.2.2 Implicit Cursor Attributes

Implicit Cursor Attributes	Meaning
SQL%ROWCOUNT	Number of records affected by the most recent SQL statement. Returns how many are there in the result set of the SQL query.
SQL%FOUND	Returns TRUE if the most recent SQL query affects at least one row
SQL%NOTFOUND	Returns TRUE if the most recent SQL query does not affect any rows.

SQL%ISOPEN

Automatically opened by Oracle engine and closes immediately after the query is executed. Hence the value of the cursor is always FALSE.

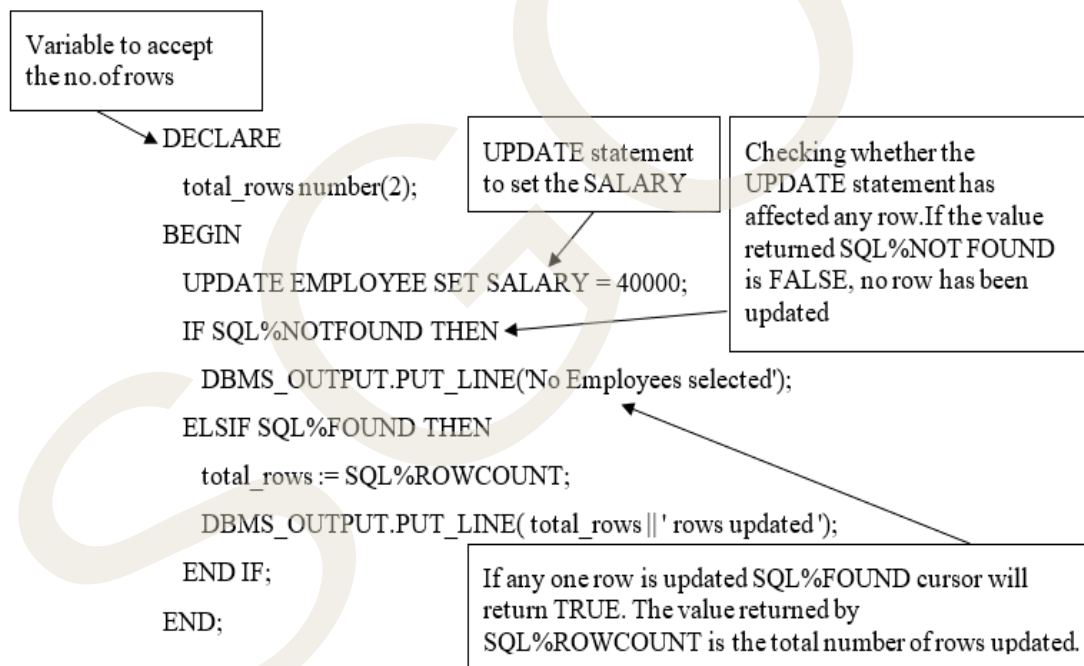
Example: Consider the table

SELECT * FROM EMPLOYEE;

Table 3.2.3 Employee Data

EMP NO	EMP NAME	DESIGNATION	STATE	SALARY	DEPT NO
E0001	JACOB	SYSADMIN	MAHARASHTRA	16500	D0001
E0002	JOHN	MANAGER	TAMILNADU	32000	D0002
E0003	ABRAHAM	DEVELOPER	KARNATAKA	26500	D0002
E0004	JOY	QA	TAMILNADU	19560	D0001
E0005	CHRISTY	SECRETARY	MAHARASHTRA	13200	D0003

Query: Update the salary of an employee to 40000



OUTPUT

5 rows updated

PL/SQL procedure successfully completed

3.2.3 Explicit Cursor

An Explicit Cursor is a cursor that is manually defined and controlled by the programmer in PL/SQL to handle query results requiring row-by-row processing. Unlike implicit cursors, explicit cursors provide more control and are suitable for scenarios where specific operations need to be performed on each row of the result set.

To use an explicit cursor, it must be declared, opened, fetched, and closed explicitly. The cursor declaration specifies the SQL query, while the fetch operation retrieves rows one at a time into PL/SQL variables. By closing the cursor, resources are released. Explicit cursors are particularly useful when working with complex queries or when conditional logic needs to be applied to each row of the result set.

- ◆ User defined cursor.
- ◆ The programmer controls the operations in the cursor.
- ◆ Operations in explicit cursor are:
 - Declaring the cursor.
 - Opening the cursor.
 - Fetching the records from the cursor.
 - Closing the cursor.

We will be using the following CUSTOMER table for explaining the explicit cursor.

Table 3.2.3 Customer Table

ID	NAME	AGE	CITY	SALARY
1	Ramesh	32	Ahmedabad	2500.00
2	Sajith	25	Delhi	2000.00
3	George	23	Kota	2500.00
4	John	25	Mumbai	7000.00
5	Mariya	27	Bhopal	9000.00
6	Saju	22	TVM	5000.00

3.2.4 Declaring the cursor

To use an Explicit Cursor, it must first be declared in the DECLARE section of the PL/SQL block. Declaring a cursor informs the Oracle engine about the name of the cursor and the SQL query it will execute. However, at this stage, the cursor is not yet opened or executed; it simply reserves a placeholder for the query.

The syntax for declaring a cursor is as follows:

CURSOR CursorName IS [SELECT statement]; OR

CURSOR cursor_name IS sql_query;

For example:

```
CURSOR emp_cursor IS SELECT emp_id, salary FROM employees WHERE  
department_id = 10;
```

This declaration sets up the cursor `emp_cursor` to retrieve employee IDs and salaries for employees in department 10.

Example:

```
CURSOR CUR_CUST IS SELECT ID, NAME, CITY FROM CUSTOMER;
```

3.2.5 Opening the cursor

In PL/SQL, cursors are opened in the **execution** or **exception** section of the block. Opening a cursor allocates memory for fetching the rows returned by the associated SQL query. At this stage, the SQL query specified during the cursor declaration is executed, and the result set is prepared for row-by-row processing. However, if an attempt is made to open a cursor that is already open, the runtime exception `CURSOR_ALREADY_OPEN` will be raised. Proper management of cursor states is essential to avoid such exceptions and ensure efficient execution.

Syntax:

```
OPEN      Cursor_Name;
```

Example:

```
OPEN CUR_CUST;
```

3.2.5.1 Fetching the record from the cursor

- ◆ Retrieves the rows into the memory variables, one row at a time.
- ◆ Each time a row is fetched, the cursor pointer is advanced to the next row in the Active Data Set.
- ◆ Fetching an unopened row will throw a runtime exception.
- ◆ If there are more than one is to be processed, the fetch statement has to be written in the LOOP...END LOOP.

Syntax:

```
FETCH CursorName INTO variable1,variable2...;
```

The variables have to be declared in the declaration section. After fetching the values to PL/SQL variables, we can process it.

Example:

```
FETCH CUR_CUST INTO v_id, v_name, v_city;
```

The above statement fetches the values of ID, NAME and CITY columns of the CUSTOMER table into the PL/SQL variables `v_id`, `v_name` and `v_city`.

3.2.6 Closing the cursor

- ◆ Disables the cursor.
- ◆ Memory area allocated is released.
- ◆ Closing an unopened cursor will throw a runtime exception.

Syntax:

```
CLOSE    Cursor Name;
```

Example:

```
CLOSE CUR_CUST;
```

3.2.7 PL/SQL Cursor with LOOP

Using a cursor with a loop in PL/SQL allows for row-by-row processing of the result set returned by an explicit cursor. This approach is commonly used when operations need to be performed on each row individually. The process involves declaring a cursor, opening it, and iterating through its rows in a loop until all rows are processed.

First, the cursor is declared in the DECLARE section, specifying the SQL query to fetch data. In the BEGIN section, the cursor is explicitly opened, allocating memory for its result set. Inside the loop, each row is fetched into a record variable matching the structure of the cursor. The loop continues to process rows until no more rows are available, as indicated by the cursor attribute %NOTFOUND.

Finally, the cursor is closed to release resources. Proper cursor management ensures efficient memory usage and avoids runtime errors. Using a cursor with a loop is particularly helpful in scenarios where conditional logic or row-specific operations are required.

1. Declaring variables using anchored declaration. The data type of columns in the database table will be

```
DECLARE
v_id CUSTOMER.ID%TYPE;
v_name CUSTOMER.NAME%TYPE;
v_city CUSTOMER.CITY%TYPE;
CURSOR CUR_CUST IS
    SELECT ID, NAME, CITY FROM CUSTOMER;
BEGIN
    OPEN CUR_CUST;
    LOOP
        FETCH CUR_CUST INTO v_id, v_name, v_city;
        EXIT WHEN CUR_CUST%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_id || ' ' || v_name || ' ' || v_city);
    END LOOP;
    CLOSE CUR_CUST;
END;
```

2. Declaring the cursor with SELECT

3. Opening the cursor.

4. Fetching each row.

5. Displaying the values

6. The loop will exit when no

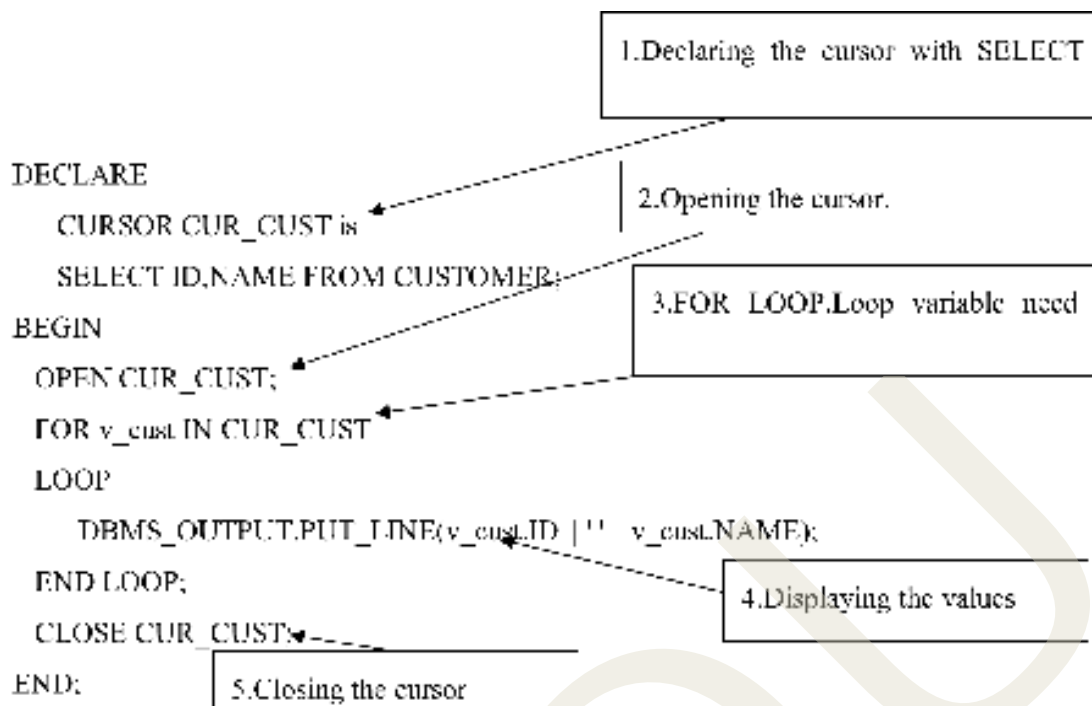
7. Closing the cursor

The above PL/SQL program will display the following output.

1. Ramesh Ahmedabad
2. Sajith Delhi
3. George Kota
4. John Mumbai
5. Mariya Bhopal
6. Saju TVM

3.2.8 PL/SQL Cursor FOR LOOP

In the previous section, we discussed the cursor with a simple loop. FOR Loop can also be used to fetch and process each row from a cursor. The previous example can be rewritten using FOR LOOP.



The above program will display the following output

```
1 Ramesh
2 Sajith
3 George
4 John
5 Mariya
6 Saju
```

PL/SQL procedure successfully completed.

In the above example, `v_cust` is a record variable.

- ◆ `v_cust` is the index of FOR LOOP.
- ◆ FOR LOOP statement declares the index implicitly as a record variable of %ROWTYPE.
- ◆ After cursor FOR LOOP statement execution ends, the record variable will be undefined.

Thus CURSOR in PL/SQL can be used to process each record in the database one by one.

Recap

- ◆ A cursor is a pointer used to manage and process multiple rows of data from a query.
- ◆ **Implicit Cursor:** Automatically created by Oracle for simple queries like SELECT INTO, INSERT, and UPDATE.
- ◆ **Explicit Cursor:** Manually declared by the programmer for complex queries requiring row-by-row processing.
- ◆ Implicit cursors are automatically managed by Oracle for simple SQL operations.
- ◆ Explicit cursors require the programmer to declare, open, fetch, and close them.
- ◆ **Declare** a cursor with a query.
- ◆ **Open** the cursor to allocate memory for fetching rows.
- ◆ **Fetch** rows one at a time.
- ◆ **Close** the cursor to release memory after processing.
- ◆ **Cursor with LOOP:** Used to fetch and process rows one by one.
- ◆ **Cursor FOR LOOP:** A simpler method to process rows with automatic fetching and looping.
- ◆ **Example of Cursor FOR LOOP:** Fetches and processes rows without manually managing the fetch logic.
- ◆ **Implicit Cursor:** Suitable for simple operations and auto-managed by Oracle.
- ◆ **Explicit Cursor:** Suitable for complex operations and manually controlled by the programmer.
- ◆ **Cursor with Loop:** Useful for row-by-row processing, especially with explicit cursors.

Objective Type Questions

1. What is a cursor in PL/SQL?
2. Which type of cursor is automatically created by Oracle for simple SQL operations?

3. What is the type of cursor that requires the programmer to manually declare and control it?
4. What operation does the OPEN statement perform on a cursor?
5. What is used to fetch rows one at a time from a cursor?
6. Which cursor type is ideal for simple SQL operations like SELECT INTO, INSERT, UPDATE?
7. Which statement is used to release memory allocated for a cursor?
8. What keyword is used to declare a cursor in PL/SQL?
9. Which cursor loop automatically fetches and processes rows?
10. What is the main benefit of using an explicit cursor?

Answers to Objective Type Questions

1. Pointer
2. Implicit Cursor
3. Explicit Cursor
4. Allocates memory for the cursor
5. FETCH
6. Implicit Cursor
7. CLOSE
8. CURSOR
9. FOR LOOP
10. Row-by-row processing

Assignments

1. Write a PL/SQL block to declare a cursor that retrieves employee names and their salaries.
2. Create a PL/SQL block to fetch and display the first employee's name from a cursor.

3. Use a cursor FOR LOOP to display the names of employees in department 20.
4. Write a PL/SQL block to update the salary of employees whose salary is less than 5000 by 10%.
5. Write a PL/SQL block to fetch and display employee details until no more rows are available using a cursor.

Suggested Reading

1. Bayross, I. (n.d.). *SQL, PL/SQL: The programming language of Oracle* (3rd ed.). BPB Publications.
2. Feuerstein, S. (n.d.). *Oracle PL/SQL programming*. O'Reilly Media.
3. McDonald, C., Katz, C., Beck, C., Kallman, J. R., & Knox, D. C. (n.d.). *Mastering Oracle PL/SQL: Practical solutions*. Apress Media.
4. Oracle Tutorial. (n.d.). PL/SQL tutorial. Retrieved March 5, 2025, from <https://www.oracletutorial.com/plsql-tutorial/>

Reference

1. tutorialspoint.com/plsql_cursor.htm
2. [geeksforgeeks.org/cursors-in-pl-sql/](https://www.geeksforgeeks.org/cursors-in-pl-sql/)
3. mygreatlearning.com/plsql/tutorials/pl-sql-cursors

Unit 3

Procedures and Functions

Learning Outcomes

Studying this unit will enable the student to:

- ◆ define a PL/SQL procedure
- ◆ describe the process of creating a procedure within a PL/SQL block
- ◆ identify the steps involved in writing a PL/SQL procedure
- ◆ explain how to compile and execute a PL/SQL procedure

Prerequisites

Before developing PL/SQL procedures and functions, a solid understanding of SQL is required. This includes knowledge of DML (Data Manipulation Language) operations like INSERT, UPDATE, and DELETE, as well as DDL (Data Definition Language) commands like CREATE, ALTER, and DROP. Familiarity with PL/SQL programming concepts such as variables, loops, conditional statements, and exception handling is also essential. Additionally, developers should understand database objects such as tables, views, sequences, and indexes to design efficient subprograms.

Another important prerequisite is knowledge of cursors, both implicit and explicit, to handle multiple rows of data efficiently. Since PL/SQL procedures and functions often interact with large datasets, understanding transaction management through COMMIT, ROLLBACK, and SAVEPOINT is crucial. Furthermore, users must have the necessary database privileges to create, modify, and execute stored procedures and functions. Performance tuning skills, including indexing strategies and query optimization, can further enhance the efficiency of these subprograms.

In real-life applications, PL/SQL procedures and functions play a key role in automating business processes. For instance, in a banking system, a stored procedure can calculate interest on savings accounts automatically. In e-commerce, functions are used to determine discounts dynamically during the checkout process. Similarly, in healthcare, PL/SQL procedures help schedule patient appointments and update medical records efficiently. These applications demonstrate how PL/SQL enhances automation, improves performance, and ensures data integrity across different industries.



Keywords

Procedure, Procedure header, Procedure body, Parameter mode

Discussion

In the previous units we discussed the basic concepts of PL/SQL and Cursors. Sometimes we need to execute a set statement repeatedly in various parts of a program. For example, we want to calculate the area of a circle in various parts of the program. We may write the code for calculating the area of a circle again and again as required. This may increase the number of lines in the program. In order to reduce the number of lines we can write the statements in a block and name it. Then the name of the block is called as and when required. Such a named block is called **Procedure**.

What if you have an SQL query to be executed over and over again? In such cases also the query can be saved in a stored procedure and just call the procedure to execute the query. The main advantage of a stored procedure is code reusability.

3.3.1 PL/SQL Procedure

A PL/SQL procedure is a subprogram designed to perform a specific task. It is a named block stored as a schema object in the Oracle database. Procedures are primarily used to perform actions. A PL/SQL procedure consists of three main parts:

3.3.1.1 Declarative Part

Optional: This part does not begin with the DECLARE keyword.

Content: It includes declarations for types, cursors, constants, variables, exceptions, and nested subprograms.

Scope: These declarations are local to the procedure and cease to exist after the procedure completes execution.

3.3.1.2 Executable Part

Mandatory: This part contains the statements that perform the designated action.

3.3.1.3 Exception-handling Part

Optional: This section includes code to handle runtime errors.

3.3.2 Syntax for Creating a PL/SQL Procedure

```
CREATE [OR REPLACE ] PROCEDURE procedure_name  
[(parameter_name [IN | OUT | IN OUT] type [, ...])]  
IS
```


[declaration statements]

BEGIN

[execution statements]

EXCEPTION

[exception handler]

END [procedure_name];

3.3.2.1 Procedure Header

The CREATE statement in Oracle is used to define a new procedure, specifying its name and associated logic. The optional [OR REPLACE] clause allows for modifying an existing procedure by replacing it with a new definition. The procedure can include a parameter list, where parameters are designated as IN, OUT, or INOUT to specify the direction of data flow between the procedure and the calling program.

3.3.2.2 Procedure Body

The body of a procedure contains the core executable part, where the main operations and logic are performed. To define a standalone procedure, you can use either the keyword IS or AS. The procedure's structure typically includes three key sections: the declarative part for declaring variables and constants, the execution part where the procedure's main actions take place, and the exception handling part, which is used to handle any errors that might occur during execution.

3.3.3 Executing a PL/SQL Procedure

3.3.3.1 Syntax

EXECUTE procedure_name(arguments);

or

EXEC procedure_name(arguments);

Example:

EXECUTE proc1();

EXECUTE proc_add(10,5);

3.3.3.2 Examples

Example 1: The following example displays the string “HELLO WORLD”.

CREATE OR REPLACE PROCEDURE proc1

AS

BEGIN

```
DBMS_OUTPUT.PUT_LINE('Hello World!');  
END;
```

When the above code is executed using the SQL prompt, it will produce the following result:

Procedure created

To execute the above procedure can be called using EXECUTE keyword.

```
EXECUTE proc1
```

The above call will display the following output

Hello World

PL/SQL procedure successfully completed.

A procedure can also be called from a PL/SQL block.

The above call will display the following output

Hello World

PL/SQL procedure successfully completed.

Example 2: Create a PL/SQL procedure that takes emp_id as input and delete the details of particular employee.

```
CREATE TABLE employees (  
    emp_id NUMBER PRIMARY KEY,  
    name VARCHAR2(50),  
    salary NUMBER  
);  
INSERT INTO employees VALUES (101, 'John Doe', 50000);  
INSERT INTO employees VALUES (102, 'Jane Smith', 60000);  
COMMIT;  
select * from employees;  
  
CREATE OR REPLACE PROCEDURE delete_employee(p_emp_id IN NUMBER)  
IS  
BEGIN  
    DELETE FROM employees WHERE emp_id = p_emp_id;  
  
    -- Check if any row was deleted  
    IF SQL%ROWCOUNT = 0 THEN  
        DBMS_OUTPUT.PUT_LINE('No record found with emp_id: ' || p_emp_id);  
    ELSE  
        DBMS_OUTPUT.PUT_LINE('Record with emp_id ' || p_emp_id || ' deleted.');    END IF;  
  
    COMMIT; -- Commit the transaction  
EXCEPTION  
    WHEN OTHERS THEN  
        ROLLBACK; -- Rollback in case of an error  
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);  
END delete_employee;  
/  
BEGIN  
    delete_employee(101);  
END;  
/  
  
select * from employees;
```

Output:

EMP_ID	NAME	SALARY
101	John Doe	50000
102	Jane Smith	60000

Record with emp_id 101 deleted.

EMP_ID	NAME	SALARY
102	Jane Smith	60000

Query:

```
CREATE OR REPLACE PROCEDURE CUST_DELETE(v_id CUSTOMER.  
ID%TYPE)
```

```
IS
```

```
BEGIN
```

```
DELETE FROM CUSTOMER WHERE ID = v_id;
```

```
END;
```

The above procedure can be executed as follows

```
EXECUTE CUST_DELETE(1)
```

The output will be

PL/SQL procedure successfully completed.

3.3.4 Parameter Modes in Procedures

In Oracle procedures, the parameter mode defines how a parameter can be used, whether it can be read, written to, or both. There are three distinct parameter modes: IN, OUT, and INOUT. Each mode serves a specific purpose in the procedure's functionality.

3.3.4.1 IN Mode

The IN mode is used for read-only parameters. Parameters in this mode are treated as constants within the procedure, which means that their values cannot be altered during execution. This mode is particularly useful for passing input data to the procedure. By default, all parameters are assumed to be in IN mode if no specific mode is declared. IN mode parameters are passed by reference, allowing the procedure to reference the original data without making a copy.

Example

```
CREATE OR REPLACE PROCEDURE DISPLAY_EMPLOYEE_
```



```

NAME(EMPLOYEE_ID IN EMPLOYEES.EMPLOYEE_ID%TYPE)
IS
    EMPLOYEE_NAME EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
    SELECT FIRST_NAME
    INTO EMPLOYEE_NAME
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = DISPLAY_EMPLOYEE_NAME.EMPLOYEE_ID;
DBMS_OUTPUT.PUT_LINE('EMPLOYEE NAME: ' || EMPLOYEE_NAME);
END;

```

In this example, the `display_employee_name` procedure takes an `employee_id` as an IN parameter. The procedure retrieves the corresponding `first_name` from the `employees` table and prints it. Since `employee_id` is an IN parameter, its value is passed to the procedure but cannot be modified within the procedure.

3.3.4.2 OUT Mode

The OUT mode is designed for parameters that need to return values back to the calling program. Unlike IN mode, OUT mode parameters act like variables inside the procedure, allowing their values to be assigned and modified. It is important to note that the actual parameter provided in the calling program must be a variable, as it will store the returned value. OUT parameters are passed by value, meaning the procedure works with a copy of the value, which is then returned to the caller.

Example

```

CREATE OR REPLACE PROCEDURE GET_EMPLOYEE_NAME(
    EMPLOYEE_ID IN EMPLOYEES.EMPLOYEE_ID%TYPE,
    EMP_NAME OUT EMPLOYEES.FIRST_NAME%TYPE
)
IS
BEGIN
    EMPLOYEE_ID
    SELECT FIRST_NAME
    INTO EMP_NAME
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = EMPLOYEE_ID;

```

END;

The GET_EMPLOYEE_NAME procedure demonstrates the use of IN and OUT parameters in PL/SQL. The EMPLOYEE_ID is an IN parameter that specifies the employee's ID, while EMP_NAME is an OUT parameter used to return the employee's first name. Inside the procedure, a SELECT INTO statement retrieves the FIRST_NAME of the employee with the given EMPLOYEE_ID and assigns it to EMP_NAME. If no matching employee is found, the exception handler assigns 'Not Found' to EMP_NAME. In the calling block, a variable V_EMP_NAME is declared to capture the output. The procedure is called with an employee ID of 100, and upon execution, V_EMP_NAME holds the employee's name, which is then printed using DBMS_OUTPUT.PUT_LINE. This example effectively shows how an OUT parameter is used to return data from a procedure to the caller.

3.3.4.3 INOUT Mode

The INOUT mode combines the functionalities of both IN and OUT modes. It allows a parameter to both receive an initial value from the calling program and return an updated value after the procedure executes. Parameters in INOUT mode can be read and modified within the procedure, making them versatile for scenarios where input data needs to be processed and returned. Like OUT mode, the actual parameter must be a variable to capture the modified value after the procedure completes.

Example

Procedure Definition:

```
CREATE OR REPLACE PROCEDURE UPDATE_SALARY(  
  EMPLOYEE_ID IN EMPLOYEES.EMPLOYEE_ID%TYPE,  
  SALARY INOUT EMPLOYEES.SALARY%TYPE  
)  
IS  
BEGIN  
  SELECT SALARY * 1.10  
  INTO SALARY  
  FROM EMPLOYEES  
  WHERE EMPLOYEE_ID = EMPLOYEE_ID;  
END;
```

Calling the Procedure:

```
DECLARE  
  
V_EMP_ID EMPLOYEES.EMPLOYEE_ID%TYPE := 100;
```

```

V_SALARY EMPLOYEES.SALARY%TYPE;

BEGIN

    SELECT SALARY INTO V_SALARY FROM EMPLOYEES WHERE
    EMPLOYEE_ID = V_EMP_ID;

    UPDATE_SALARY(V_EMP_ID, V_SALARY);

    DBMS_OUTPUT.PUT_LINE('Updated Salary: ' || V_SALARY);

END;

```

The INOUT mode in PL/SQL allows a parameter to both receive an initial value from the calling program (like an IN parameter) and return an updated value after the procedure executes (like an OUT parameter). This mode is useful when input data needs to be processed and modified, such as updating values. The parameter can be read and modified within the procedure, and the actual parameter passed must be a variable to capture the modified value after execution. In the provided example, the procedure UPDATE_SALARY takes an EMPLOYEE_ID as an input to identify the employee and an INOUT parameter SALARY that starts with the employee's current salary and returns the updated salary after applying a 10% increase, which is then displayed in the calling block.

3.3.5 Example Questions

Example 1:

Write a PL/SQL program to find the smallest among two numbers using procedure.

```

DECLARE
    x NUMBER;
    y NUMBER;
    z NUMBER;
    PROCEDURE proc_Min(a IN NUMBER, b IN NUMBER, c OUT NUMBER) IS
    BEGIN
        IF a < b THEN
            c := a;
        ELSE
            c := b;
        END IF;
    END proc_Min;

BEGIN
    x := 40;
    y := 35;
    proc_Min(x, y, z);
    DBMS_OUTPUT.PUT_LINE('Minimum of (40, 35) : ' || z);
END;
/

```

3.3.6 Deleting a Standalone Procedure

A standalone procedure can be deleted using the DROP keyword.

Syntax:

```
DROP procedure_name;
```

Example:

```
DROP proc1;
```

In this way you can create named procedures and reuse the code as and when needed.

Recap

- ◆ PL/SQL procedures are named blocks of code that execute a specific task and can be stored for reuse.
- ◆ They help in modular programming by allowing code to be divided into smaller, manageable parts.
- ◆ Procedures can take input parameters, process data, and return results, making them flexible for various applications.
- ◆ Functions in PL/SQL are similar to procedures but must return a single value.
- ◆ Both procedures and functions enhance code reusability, maintainability, and performance optimization.
- ◆ Exception handling in PL/SQL ensures that runtime errors are managed effectively without disrupting program execution.
- ◆ Triggers are special types of procedures that automatically execute in response to specific database events.
- ◆ PL/SQL procedures are widely used in real-world applications like banking, e-commerce, and healthcare for data processing.
- ◆ Understanding cursors and transactions is crucial for handling multiple records and maintaining database consistency.
- ◆ The syntax for creating a PL/SQL procedure starts with CREATE [OR REPLACE] PROCEDURE procedure_name followed by optional parameters.
- ◆ The procedure body consists of three main parts: the declarative section (optional), the executable section (mandatory), and the exception-handling section (optional).
- ◆ Parameters in a procedure can be IN, OUT, or INOUT, specifying whether they receive, return, or both receive and return values.
- ◆ The BEGIN and END keywords define the executable part of the procedure where the main logic is implemented.
- ◆ Overall, mastering PL/SQL procedures and functions allows developers to build efficient, secure, and scalable database applications.

- ◆ Stored procedures improve database performance by reducing network traffic and precompiling SQL statements.
- ◆ Proper use of PL/SQL procedures and functions ensures data integrity and enhances system security.
- ◆ The DROP PROCEDURE procedure_name; command is used to delete a procedure from the database when it is no longer needed.

Objective Type Questions

1. Which keyword is used to create a procedure in PL/SQL?
2. Which section of a PL/SQL procedure is mandatory?
3. What is the purpose of the IN parameter in a procedure?
4. What is the default mode of parameters in a PL/SQL procedure?
5. Where is a PL/SQL procedure stored?
6. How can a procedure return multiple values?
7. What happens when a stored procedure is compiled in PL/SQL?
8. Which keyword is used to define a variable inside a PL/SQL procedure?
9. What is the purpose of the RETURN statement in a procedure?
10. Which of the following statements is used to remove a stored procedure from the database?
11. Which statement is used inside a procedure to stop its execution?
12. What happens if a stored procedure is created with the same name as an existing one?
13. What is the scope of variables declared in a procedure?
14. Which SQL clause is used to call a stored procedure within another procedure?

Answers to Objective Type Questions

1. CREATE PROCEDURE
2. Executable
3. It allows input from the calling program
4. IN
5. In the database
6. By using OUT parameters
7. It is stored in the database for future execution
8. DECLARE
9. It exits the procedure.
10. DROP PROCEDURE procedure_name.
11. RETURN
12. The new procedure overwrites the old one if CREATE OR REPLACE is used
13. They are available only within the procedure
14. CALL

Assignments

1. Write a PL/SQL procedure named calculate_bonus that takes an employee's ID as input and calculates a 10% bonus based on their current salary. The procedure should update the employee's bonus in the employees table.
2. Develop a PL/SQL function named get_department_name that accepts a department ID as input and returns the name of the department from the departments table. Include appropriate error handling for invalid department IDs.
3. Create a PL/SQL procedure named update_salary that increases an employee's salary by a percentage specified by a function named calculate_increment. The function should take the current salary and a percentage as parameters and return the new salary.

Suggested Reading

1. Bayross, I. (n.d.). *SQL, PL/SQL: The programming language of Oracle* (3rd ed.). BPB Publications.
2. Feuerstein, S. (n.d.). *Oracle PL/SQL programming*. O'Reilly Media.
3. McDonald, C., Katz, C., Beck, C., Kallman, J. R., & Knox, D. C. (n.d.). *Mastering Oracle PL/SQL: Practical solutions*. Apress.
4. Oracle Tutorial. (n.d.). PL/SQL tutorial. Retrieved March 5, 2025, from <https://www.oracletutorial.com/plsql-tutorial/>

Reference

1. Feuerstein, S. (2005). *Oracle PL/SQL programming*. O'Reilly Media.
2. Rosenzweig, B., & Silvestrova, E. (2016). *Oracle PL/SQL by example*. Pearson.
3. Pribyl, B., & Feuerstein, S. (2003). *Learning Oracle PL/SQL*. O'Reilly Media.
4. Feuerstein, S. (2014). *Oracle PL/SQL for dummies*. Wiley.

Unit 4

Triggers

Learning Outcomes

Studying this unit will enable the student to:

- ◆ define database triggers as stored programs
- ◆ identify and explain the use of triggers for enforcing constraints
- ◆ apply triggers to solve real-world database problems
- ◆ write triggers to automatically update related table columns
- ◆ use triggers to prevent unwanted data modification

Prerequisites

Building upon your established expertise in PL/SQL procedures and functions, you're now poised to explore the realm of triggers. Think of your prior knowledge as mastering the art of instructing a skilled assistant: procedures allow you to give specific, step-by-step commands, like generating a monthly sales report or processing a customer refund, while functions provide the ability to ask the assistant to calculate a value, such as determining a customer's total purchase amount or applying a discount. Now, triggers introduce a level of automation, allowing the database itself to become a self-regulating system. Imagine a real-world scenario where you need to maintain an audit log of all changes to customer addresses. Instead of relying on someone to manually record these changes every time an update occurs, a trigger can automatically capture the old and new addresses, along with the timestamp and user who made the change, ensuring data integrity and compliance.

Triggers essentially act as automatic guardians, ensuring that predefined actions are taken whenever specific events occur within your database. Consider an inventory management system: every time a new product is added (INSERT), a trigger could automatically update the total number of items in stock. Similarly, when a product's price is updated (UPDATE), a trigger could notify the sales department of the change. Or, when a product is removed (DELETE), a trigger could archive the product's data for historical analysis. These automated actions, triggered by data modifications, ensure that business rules are consistently enforced and that related data is always synchronized. Just as your assistant, armed with your procedures and functions, efficiently handles tasks upon your command, triggers enable your database to automatically respond to



changes, ensuring data consistency and automating repetitive tasks without manual intervention. By leveraging your existing knowledge of procedures and functions, you'll find that triggers seamlessly extend your ability to automate and streamline your database operations.

Keywords

Trigger, CREATE TRIGGER, row-level trigger, statement-level trigger

Discussion

3.4.1 Introduction to Triggers

In the previous unit, we discussed PL/SQL procedures, which allow for code reusability. Similar to procedures, triggers enable multiple applications to execute a set of SQL statements, reducing redundant code when multiple applications need to perform the same database operation.

For instance, consider an online shopping scenario where a business rule restricts a customer from ordering more than one bottle of perfume at a time. If an attempt is made to order more than one, a trigger can enforce this rule by displaying a warning message.

3.4.2 Definition and Purpose of Triggers

- ◆ Triggers are stored programs that are automatically executed or "fired" when a specific event occurs in the database.
- ◆ Triggers are typically written to respond to SQL commands such as INSERT, DELETE, or UPDATE.

3.4.3 Usages of Triggers

Triggers are used for various purposes, including:

- ◆ Enforcing **UNIQUE**, **CHECK**, and **NOT NULL** constraints.
- ◆ Preventing **invalid transactions**.
- ◆ Generating values for **derived attributes**. For example, while inserting the date of birth of an employee, the age column can be automatically calculated and filled.
- ◆ **Auditing** sensitive data.

Syntax for creating triggers:

```
CREATE [OR REPLACE ] TRIGGER trigger_name  
{BEFORE | AFTER}
```

ON table_name
[FOR EACH ROW]
[FOLLOWS | PRECEDES another_trigger]
[ENABLE / DISABLE]
[WHEN condition]
DECLARE
Declaration-statements
BEGIN
Executable-statements
EXCEPTION
Exception-handling-statements

3.4.3.1 Trigger Header

The following part is the trigger header.

CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER}
ON table_name
[FOR EACH ROW]
[FOLLOWS | PRECEDES another_trigger]
[ENABLE / DISABLE]
[WHEN condition]

3.4.3.2 Trigger Body

The following part illustrates trigger body

DECLARE
Declaration-statements
BEGIN
Executable-statements
EXCEPTION
Exception-handling-statements
END;

◆ CREATE OR REPLACE

- Keyword for creating a trigger.
- OR REPLACE is used to modify an existing trigger. Optional. But it is recommended.

Example:

```
CREATE OR REPLACE trg1_example
```

The above statement creates a trigger named trg1_example. If such a trigger exists, it will replace the existing trigger.

trigger_name

- ◆ specifies the name of the trigger.

BEFORE | AFTER

- ◆ Specifies when the trigger should be fired.
- ◆ BEFORE- the trigger is fired before INSERT, DELETE, UPDATE statement.
- ◆ AFTER-the trigger is fired after INSERT, DELETE, AND UPDATE statement.

ON table_name

- ◆ Name of the table on which the trigger is associated.

FOR EACH ROW

- ◆ Specifies a row-level trigger.
- ◆ For each row inserted or deleted or updated, the trigger will be fired.
- ◆ **Statement-level trigger**-Fired only once regardless of number of rows affected by the triggering event.
- ◆ If the clause FOR EACH ROW is not specified, the trigger will be created as a statement-level trigger.

ENABLE | DISABLE

- ◆ Specifies whether the trigger is in ENABLE or DISABLE state.
- ◆ If a trigger is in DISABLE state, it will not be fired.
- ◆ Default state is ENABLE.

FOLLOWS | PRECEDES another_trigger

- ◆ Multiple triggers can be fired for each trigger.

- ◆ Specify firing sequence using PRECEDES or FOLLOWS keyword

3.4.3.3 Dropping Triggers

A trigger can be dropped using DROP TRIGGER command

Syntax

```
DROP TRIGGER trigger_name;
```

Example

```
DROP TRIGGER trg1_example;
```

Example:The following program illustrates a trigger

Consider the CUSTOMER table as shown below.

Table 3.4.1 Customer table

ID	NAME	AGE	CITY	SALARY
1	Ramesh	32	Ahmedabad	2500.00
2	Sajith	25	Delhi	2000.00
3	George	23	Kota	2500.00
4	John	25	Mumbai	7000.00
5	Mariya	27	Bhopal	9000.00
6	Saju	22	TVM	5000.00

Our aim is to create a trigger that is fired during the insertion of a new record.

```
CREATE OR REPLACE TRIGGER trg_display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON CUSTOMER
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
diff_sal number;
BEGIN
diff_sal := :NEW.salary - :OLD.salary;
dbms_output.put_line('Old salary: ' || :OLD.salary);
dbms_output.put_line('New salary: ' || :NEW.salary);
dbms_output.put_line('Salary difference: ' || diff_sal);
END;
```

The above code, when executed gives the following output.

Trigger created.

The trigger created will be fired each time a row is inserted into the table.

Now, let us insert a row into the table using an INSERT statement.

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Joseph', 22, 'HP', 7500.00 );
```

When a record is inserted, the trigger `trg_display_salary_changes` is fired and the following message is displayed.

Old salary:

New salary: 7500

Salary difference:

Recap

- ◆ Triggers are stored programs that execute automatically when a specific event occurs in the database.
- ◆ Triggers help to reduce redundant code and ensure consistency across multiple applications.
- ◆ Usages of Triggers

Enforcing constraints (UNIQUE, CHECK, NOT NULL).

Preventing invalid transactions.

Automatically generating values for derived attributes (e.g., calculating age from date of birth).

Auditing changes to sensitive data.

- ◆ Syntax to create triggers:

```
CREATE [OR REPLACE] TRIGGER trigger_name
```

```
BEFORE | AFTER} on a specified table.
```

```
Optional: [FOR EACH ROW], [ENABLE/DISABLE], [WHEN
condition], [FOLLOWS | PRECEDES another_trigger]
```

◆ **Trigger Header:**

Includes trigger name, timing (BEFORE/AFTER), table name, optional conditions, and **enable/disable options**.

◆ **Trigger Body:**

Includes declaration, executable, and exception-handling statements.

Key Syntax Elements

`CREATE OR REPLACE TRIGGER trigger_name:` Creates or replaces a trigger.

`BEFORE | AFTER:` Determines when the trigger fires.

`ON table_name:` Specifies the table associated with the trigger.

`FOR EACH ROW:` Defines a row-level trigger (executes once per affected row).

`ENABLE | DISABLE:` Controls whether the trigger is active or inactive.

`FOLLOWS | PRECEDES another_trigger:` Specifies the execution order of multiple triggers.

- ◆ Triggers can be dropped using `DROP TRIGGER`.
- ◆ Triggers help maintain data integrity and automate database operations efficiently.
- ◆ Triggers are an essential tool in PL/SQL programming for enforcing business rules and auditing data changes.

Objective Type Questions

1. Which trigger type executes before a DML operation?
2. Which trigger is used for modifiable views?
3. What keyword defines a trigger?
4. A trigger that triggers another trigger is:

5. Triggers are stored in:
6. Triggers can prevent:
7. How many triggers can be created per table for the same event and timing in MySQL?
8. Can a trigger call a stored procedure in MySQL?
9. How many types of triggers does MySQL support?
10. What is the maximum number of triggers a single table can have in MySQL?
11. What is the default activation mode of a MySQL trigger?
12. What will happen if you try to use the OLD keyword in a BEFORE INSERT trigger?
13. Which MySQL statement is used to remove a trigger?

Answers to Objective Type Questions

1. BEFORE
2. INSTEAD OF
3. TRIGGER
4. Cascading trigger
5. Data dictionary
6. Unauthorized modifications
7. Only one
8. Yes
9. 6
10. 12 (6 BEFORE and 6 AFTER triggers per event type)
11. FOR EACH ROW
12. It will cause a syntax error
13. DROP TRIGGER

Assignments

1. Write a PL/SQL trigger named `log_employee_changes` that activates after any UPDATE operation on the `employees` table. The trigger should insert a record into an `employee_audit` table, logging the employee ID, old salary, new salary, and the update timestamp.
2. Create a trigger named `prevent_negative_salary` on the `employees` table that fires before any INSERT or UPDATE. The trigger should ensure that the salary value cannot be negative. If a negative salary is detected, raise a custom exception with an appropriate error message.
3. Develop a trigger named `delete_department_employees` that activates before a DELETE operation on the `departments` table. The trigger should automatically delete all employees associated with the department being removed, ensuring data consistency.
4. Write a trigger named `update_inventory_stock` that fires after an INSERT or UPDATE on the `order_details` table. The trigger should automatically update the `stock_quantity` in the `inventory` table based on the quantity of items ordered.

Suggested Reading

1. Bayross, I. (n.d.). *SQL, PL/SQL: The programming language of Oracle* (3rd ed.). BPB Publications.
2. Feuerstein, S. (n.d.). *Oracle PL/SQL programming*. O'Reilly Media.
3. McDonald, C., Katz, C., Beck, C., Kallman, J. R., & Knox, D. C. (n.d.). *Mastering Oracle PL/SQL: Practical solutions*. Apress.
4. Oracle Tutorial. (n.d.). *PL/SQL tutorial*. Retrieved March 5, 2025, from <https://www.oracletutorial.com/plsql-tutorial/>

Reference

1. [geeksforgeeks.org/sql-trigger-student-Database/](https://www.geeksforgeeks.org/sql-trigger-student-Database/)
2. archive.nptel.ac.in/course/106/104/106104135/



Transaction Management

Unit 1

Introduction to Transaction Management

Learning Outcomes

The learner will be able to:

Studying this unit will enable the student to:

- ◆ understand the concept of transaction in the database
- ◆ describe the properties of transactions
- ◆ understand the different states of transaction
- ◆ familiarize with concurrency control mechanisms

Prerequisites

When you withdraw money from an ATM, book a movie ticket online, or place an order on an e-commerce website, you're unknowingly relying on a crucial concept in databases—transactions. A transaction is a set of operations that must be executed completely or not at all. Imagine transferring money from one account to another; if the amount is deducted from one account but not credited to the other due to a failure, it could lead to serious errors. To prevent such issues, databases follow a principle called ACID properties—Atomicity, Consistency, Isolation, and Durability—to ensure that transactions are processed reliably.

But what happens if there is a power failure while booking a ticket? What if two people try to book the last available seat at the same time? These real-world scenarios highlight the importance of Transaction Management in a Database Management System (DBMS). Transaction management ensures that all database operations are carried out smoothly and securely. If an error occurs in the middle of a transaction, the system ensures that the changes are either completely applied or completely undone, preventing data inconsistencies. This is what makes banking systems, online shopping, and even social media platforms function seamlessly without errors or loss of critical information.

In this chapter, we will explore how databases handle transactions, ensure data consistency, and manage concurrent user access efficiently. You will learn about key concepts such as commit, rollback, concurrency control, and recovery techniques. By the end, you will understand how databases maintain integrity even when multiple users interact with them simultaneously. Are you ready to explore the hidden engine that keeps digital transactions safe and reliable? Let's get started!



Keywords

Transaction, Commit, Abort, Rollback, System Log, Atomicity, Concurrency, Isolation, Durability

Discussion

Raju wanted to withdraw some amount of money from his bank account. He went to the ATM counter, entered the pin and the amount to be withdrawn. The machine dispensed the amount and Raju collected the cash. The amount withdrawn was debited from his account and the new balance is displayed on the ATM screen. This is an example of everyday activity that we are familiar with. Similarly, Raju can deposit money in his bank account or transfer the amount from his account to somebody else's account.

Today, many people prefer shopping online for their favorite items. Similarly, when booking tickets for flights or trains, we use an online reservation system. All these activities rely on accessing and managing a database. Each of these actions involves processing data in the database, and this collection of actions is known as a transaction.

4.1.1 Transaction

Any logical operation or set of operations performed on the data within a database is known as a transaction. These operations can include inserting new data, deleting existing records, or updating values within the database. The primary objective of a transaction is to ensure that the database remains accurate, consistent, and reliable, even in the event of system failures or concurrent user access.

A transaction can be executed using a specialized query language known as Structured Query Language (SQL) or embedded within an application program. Depending on its nature, a transaction can be classified into two types. A read-only transaction retrieves data from the database without making any modifications. For example, checking the balance in a bank account is a read-only operation. On the other hand, a read-write transaction involves both retrieving and modifying data. For instance, depositing or withdrawing money from a bank account requires updating the account balance in the database.

Consider a scenario where you are booking a movie ticket online. You select a seat, enter your payment details, and confirm your booking. If, for any reason, the payment fails, the system ensures that your seat is not reserved. This process of grouping related database operations together as a single unit is an example of a transaction. In this chapter, we will explore how databases manage transactions, ensuring that data remains correct, secure, and efficiently processed even when multiple users interact with the system at the same time.

4.1.2 Transaction Management

Raju went to the ATM to withdraw money from his bank account. He entered the pin and amount to be retrieved. A system crash occurred, and Raju did not get the money.

But the amount was debited from his account. The transaction has failed because Raju did not receive the money. Here the bank will revert the transaction and the amount will be credited back to his account. The transaction will also fail if Raju does not have sufficient funds in his account. A transaction may fail or may not be completed due to several reasons like system crash, unavailability of resources, network error. When a failure occurs, the transaction will be cancelled, and the database will be restored to the same previous state before the transaction is executed.

Commit

If all the operations in a transaction are completed successfully, the changes are recorded permanently in the database.

Rollback

If the transaction is not successful, the changes are not recorded in the database and the transaction will be aborted or canceled. Undo all the current operations and the database is restored to the previous state.

System Log

A record maintained by DBMS to keep track of all transactions that affect data in the database. Each entry is a record or row, which consists of:

- ◆ Transaction id, a unique identifier assigned to every transaction by DBMS.
- ◆ Operations; read or write.
- ◆ Commit or Abort.

4.1.3 Transaction States

- ◆ A **transaction state** refers to the status or condition of a transaction at any given point during its lifecycle. Fig 1 shows how a transaction state works.

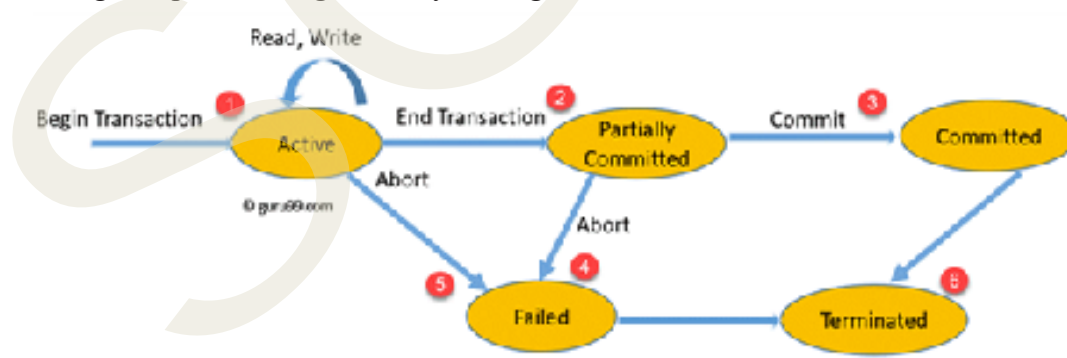


Fig. 4.1.1 States of transaction in DBMS

- ◆ **New**
Marks the beginning of a transaction
- ◆ **Active**
The transaction is executing. Operations like insertion, deletion, updation are done

in this stage. However, the changes have not been saved to the database permanently

◆ **Partially Committed**

This state happens after the last operation of the transaction is completed, but before the transaction is officially finalized in the database. The transaction is almost complete, but the system is verifying that everything is accurate before making the changes permanent.

◆ **Committed**

The transaction executed successfully, and the modifications are made permanent on the database

◆ **Failed**

The transaction failed to complete successfully because of an error or some issue (such as data inconsistency or a system crash). In this state, the transaction is not finalized, and its changes are not applied to the database.

◆ **Aborted**

The state when a transaction is rolled back due to failure, restoring the database to its previous state.

◆ **Terminated**

The state after a transaction has reached a committed or aborted state

4.1.4 Transaction Properties

- ◆ To maintain database consistency both before and after a transaction, specific properties, known as **ACID properties**, shown in Fig 2, must be followed.
- ◆ These properties are essential when multiple transactions occur at the same time

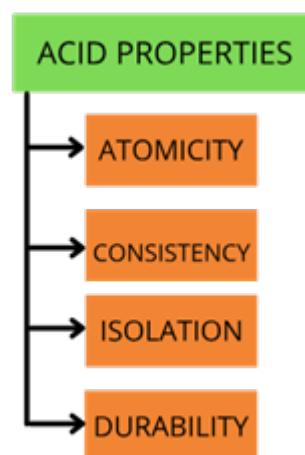


Fig. 4.1.2 ACID properties

◆ Atomicity

A transaction is atomic, meaning it is treated as a single, indivisible unit. Atomicity is also known as the 'All or nothing rule'. Either the entire transaction is completed successfully, or not executed at all in case of failure. If something goes wrong, any changes made by the transaction are rolled back. In the ATM example, If the system crashes while Raju withdraws money, Atomicity ensures the transaction is either fully completed (money withdrawn and debited) or fully canceled (no money withdrawn and no change in the account).

◆ Consistency

Refers to the correctness of data. Data integrity must be maintained, meaning that any modifications to the database should be permanent.

Example: Raju's account balance is correct before and after he tries to withdraw money, even if something goes wrong.

◆ Isolation

Each transaction should operate independently of others.

Even if multiple transactions occur simultaneously, they should behave as if they were processed one at a time, without affecting each other.

Example: If someone else is using the ATM while Raju is, their transaction will not interfere with his, and he will not see their balance.

◆ Durability

Once a transaction is committed (successfully completed), its changes are permanent, even if the system crashes. The database ensures that the data persists and can be recovered if needed. Durability property is the responsibility of the recovery subsystem in DBMS.

Example: If Raju successfully withdraws money and the system crashes right after, his balance will still show the correct amount when the system is back online.

4.1.5 Concurrency Control

In a multi-user database system, multiple transactions often occur at the same time. If not managed properly, simultaneous transactions can lead to data inconsistencies, conflicts, or even loss of critical information. Concurrency control is a mechanism used in Database Management Systems (DBMS) to ensure that multiple transactions can execute smoothly while maintaining data consistency and integrity.

Imagine an online shopping website where multiple customers are trying to buy the last available item. If two people select the same item at the same time, who gets to buy it? Without proper management, the system might end up selling the same item to multiple users, leading to confusion and errors. Concurrency control ensures that database transactions are executed correctly and consistently, even when many users access the system at once.

To maintain accuracy and consistency, databases follow some key Concurrency Control Techniques, including:

1. **Locking Mechanisms:** Prevent multiple users from modifying the same data at the same time.
2. **Time-stamping:** Ensures that transactions are executed in the correct order.
3. **Serializability:** Ensures that transactions occur in a way that does not cause conflicts or data corruption.

4.1.5.1 Locking Mechanisms in Concurrency control

In a multi-user database system, several users or applications may try to access or modify the same data simultaneously. Without a proper control mechanism, conflicting operations can lead to data inconsistencies, incorrect results, or even system crashes. Locking mechanisms are used in a Database Management System (DBMS) to ensure that transactions are executed in a safe and controlled manner while maintaining data consistency and integrity.

How Locking Works?

When a transaction needs to access a piece of data, the DBMS places a lock on that data. A lock prevents other transactions from making conflicting changes until the current transaction completes its operation. Once the transaction finishes, the lock is released, allowing other transactions to proceed.

Locks can be categorized into two main types:

1. **Shared Lock (S-Lock):** A shared lock allows multiple transactions to read the same data simultaneously but prevents any modifications. This ensures that multiple users can safely retrieve information without causing inconsistencies. For example, in an online banking system, multiple users can check their account balances at the same time, but none of them can modify the balance while the lock is active.
2. **Exclusive Lock (X-Lock):** An exclusive lock restricts all access to a data item. When a transaction holds an exclusive lock on a data record, no other transaction can read or modify that data until the lock is released. This ensures that no other transaction interferes while the current transaction is updating or deleting data. For instance, if a user transfers money from one account to another, an exclusive lock is placed on the account balance to prevent any other transaction from making changes until the transfer is complete.

4.1.5.2 Time-Stamping in DBMS

Time-stamping is a concurrency control mechanism that assigns a unique timestamp to each transaction when it begins. This timestamp determines the order in which transactions should be executed. The idea is simple: transactions that start earlier should be processed first to maintain consistency in the database.

How Does Time-Stamping Work?

Each transaction in a database is given a unique timestamp when it starts. The DBMS ensures that:

- ◆ If an older transaction (T1) conflicts with a newer one (T2), T1 gets priority. The newer transaction (T2) may have to wait or restart to avoid conflicts.
- ◆ A younger transaction (T2) cannot overwrite changes made by an older one (T1). If T1 starts first, it must finish first, ensuring a proper sequence of execution.

For example, consider two people trying to book the last available seat for a movie at the same time. If both transactions are processed without control, the system might end up selling the same seat to both users. Time-stamping prevents such conflicts by ensuring that the first user to book the seat gets priority, while the second user either waits or gets an update if the seat is unavailable. This technique ensures accuracy and prevents data corruption in high-demand applications like online banking, airline reservations, and e-commerce platforms.

Recap

- ◆ **Transaction:** A logical operation or a set of operations performed on a database. Includes inserting, deleting, or updating data.
- ◆ **Goal of Transaction:** Ensures database accuracy, consistency, and reliability, even during failures.
- ◆ **Types of Transactions:**
 - **Read-Only Transaction:** Retrieves data without modifications (e.g., checking a bank balance).
 - **Read-Write Transaction:** Retrieves and modifies data, such as depositing or withdrawing money from a bank account.

Transaction Management

- ◆ Ensures successful completion or safe rollback of transactions in case of failures.
- ◆ **Example:** If an ATM crashes while withdrawing money, the system ensures the transaction is canceled or completed properly.
- ◆ **Failure Causes:** System crash, resource unavailability, network errors, data inconsistency.
- ◆ **Database Restoration:** If a failure occurs, the database is restored to its previous state before the transaction.

Commit & Rollback

- ◆ **Commit:** Confirms and saves all completed operations in a transaction permanently.

- ◆ **Rollback (Abort):** If a failure occurs, all changes made by the transaction are undone, returning the database to its previous state.

System Log

- ◆ **Transaction ID:** Unique identifier assigned to each transaction.
- ◆ **Operations:** Tracks actions like read and write operations.
- ◆ **Commit or Abort Status:** Logs whether a transaction is successfully completed or canceled.

Transaction States

- ◆ **New:** Transaction begins.
- ◆ **Active:** Transaction is executing (insert, delete, update operations).
- ◆ **Partially Committed:** All operations are done, but verification is in progress before finalizing.
- ◆ **Committed:** Transaction successfully completed, and changes are saved.
- ◆ **Failed:** Transaction cannot be completed due to an error (e.g., system crash, insufficient funds).
- ◆ **Aborted:** Transaction is rolled back, and changes are undone.
- ◆ **Terminated:** Transaction has reached either the committed or aborted state.

Transaction Properties (ACID)

- ◆ **Atomicity:** Transaction is treated as a single, indivisible unit (All or Nothing).
- ◆ **Consistency:** Ensures database remains in a valid state before and after a transaction.
- ◆ **Isolation:** Transactions execute independently without interfering with each other.
- ◆ **Durability:** Ensures committed transactions are permanently saved, even after failures.

Concurrency Control

- ◆ **Ensures data consistency** in multi-user environments where multiple transactions run simultaneously.
- ◆ Prevents conflicts like **lost updates, dirty reads, and data inconsistencies.**
- ◆ Used in systems like **banking, e-commerce, and online ticket booking.**

Concurrency Control Techniques

- ◆ **Locking Mechanisms:** Restrict multiple transactions from modifying the same data simultaneously.
- ◆ **Timestamp Ordering:** Assigns unique timestamps to transactions to ensure correct execution order.
- ◆ **Serializability:** Ensures that multiple transactions do not interfere with each other, maintaining data consistency.

Locking Mechanisms

- ◆ **Shared Lock (S-Lock):** Multiple transactions can read the same data but cannot modify it. Example: Checking bank balance.
- ◆ **Exclusive Lock (X-Lock):** Only one transaction can access and modify the data. Example: Withdrawing money from an ATM.

Objective Type Questions

1. What is a logical operation performed on database data called?
2. Which programming language is known for its simplicity and readability?
3. What is the primary goal of a transaction in a database?
4. What is the term for a transaction that only retrieves data without modifying it?
5. What happens when all operations in a transaction are successfully completed?
6. What property ensures that a transaction is either fully completed or not executed at all?
7. Which property ensures that multiple transactions do not interfere with each other?
8. What is the full form of ACID in database transactions?
9. What is the process of grouping multiple operations into a single unit in a database?
10. What is the system record that tracks all transactions in a database called?
11. In which state does a transaction start executing?
12. Which transaction state occurs after the last operation is completed but before finalization?

13. What does a shared lock allow transactions to do with the same data?
14. What kind of lock allows only one transaction to access and modify a data item at a time?
15. What is the name of the concurrency control technique that assigns a unique identifier to each transaction to ensure proper execution order?

Answers to Objective Type Questions

1. Transaction
2. Python
3. Consistency
4. Read-only
5. Transaction
6. Atomicity
7. Isolation
8. Atomicity, Consistency, Isolation, Durability
9. New
10. System Log
11. Active
12. Exclusive (X-Lock)
13. Shared
14. Exclusive Lock
15. Time-stamping

Assignments

1. What is a transaction in a database, and why is it considered a logical unit of work?
2. Explain the importance of Atomicity in transaction management with an example.
3. What is the difference between a commit and a rollback in a database transaction?

4. How does a shared lock (S-Lock) differ from an exclusive lock (X-Lock) in concurrency control?
5. Explain how time-stamping helps maintain data consistency in a multi-user database system.

Suggested Reading

1. Elmasri, R., & Navathe, S. B. (2006). *Fundamentals of database systems* (5th ed.). Pearson.
2. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). *Database system concepts* (6th ed.). Tata McGraw-Hill.
3. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). Tata McGraw-Hill.
4. Date, C. J., Kannan, A., & Swamynathan, S. (2006). *An introduction to database systems* (8th ed.). Pearson Education.

Reference

1. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Pearson Education.
2. Gray, J., & Reuter, A. (1992). *Transaction processing: Concepts and techniques*. Morgan Kaufmann.

Unit 2

Serializability

Learning Outcomes

The learner will be able to:

- ◆ define the term schedule in the context of database transactions
- ◆ list the different types of schedules used in database management systems
- ◆ identify the conditions for a conflict serializable schedule
- ◆ explain the steps of the conflict serializability algorithm
- ◆ familiarize the three conditions for view serializability

Prerequisites

In our daily lives, we often perform multiple tasks at the same time. For example, when using a computer, we may open multiple applications, such as a web browser, a document editor, and a music player, all running together. The operating system manages these tasks by switching between them efficiently. Similarly, in a database system, multiple transactions can run at the same time, with their operations interleaved to ensure smooth processing.

In this unit, we will explore schedules—the order in which database transactions execute—and understand how they maintain consistency. You will learn about serial and non-serial schedules, how to determine if a schedule is serializable, and the methods used to check conflict serializability and view serializability. By the end of this discussion, you will see how database management systems ensure that transactions execute in a correct and efficient manner, just like how an operating system handles multiple processes effectively.

Keywords

Transaction, Schedule, Serial Schedule, Non-serial schedule, Serializable schedule, Serializability, Conflict Serializability, View Serializability



Discussion

Consider two processes, A and B, running simultaneously on a multiprogramming operating system. The operating system alternates between executing commands from process A and process B to make efficient use of the CPU. Similarly, in a DBMS, when multiple transactions run concurrently, their operations are executed in an interleaved manner. The order in which these operations are executed - (whether for processes or database transactions) - is known as a schedule. This ensures that operations are carried out efficiently, while maintaining the integrity and consistency of the database.

4.2.1 What is a Schedule?

- ◆ The order in which the operations (such as read, write, commit, etc.) of multiple transactions are executed in a database system
- ◆ The primary goal of a schedule is to ensure that concurrent transactions are executed in a way that maintains the integrity and consistency of the database
- ◆ For example, if two people, John and Sam try to withdraw money from the same ATM at the same time, the ATM handles their requests in a specific order to make sure both receive the correct amount, and the balance is updated correctly. The schedule is simply the plan that ensures these transactions happen step by step without errors.
- ◆ **There are 4 operations** in a schedule. They are:
 - ◆ read
 - ◆ write
 - ◆ commit
 - ◆ abort

Example, consider two transactions, T1 and T2 given in Figure 4.2.1 executing in an inter-leaved fashion.

T1	T2
$r_1(A)$ $w_1(A)$	$r_2(B)$ $w_2(B)$

Fig. 4.2.1 Schedule S of transactions T1 and T2

The schedule S will be as follows: S: $r_1(A)$; $w_1(A)$; $r_2(B)$; $w_2(B)$;

where $r_1(A)$ and $w_1(A)$ denote read and write operations performed by T1 $r_2(B)$ and $w_2(B)$ denote read and write operations performed by T2.

4.2.2 Types of Schedules in DBMS

There are different types of schedules based on how the operations are arranged and whether they maintain the required properties of transactions. Fig 4.2.2 shows the different types of schedules.

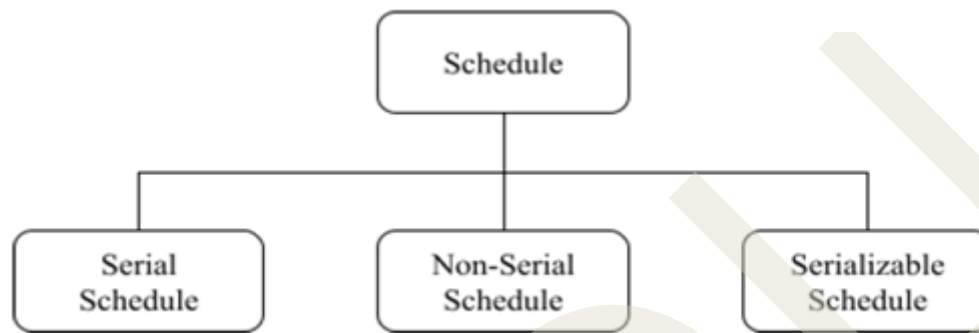


Fig. 4.2.2 Schedule Types

4.2.2.1 Serial Schedule

- ◆ Transactions are executed one after the other, without any overlap or interleaving of operations
- ◆ Operations of one transaction are executed in sequence, and once a transaction finishes, the next one begins
- ◆ Only one transaction is active at a time
- ◆ The second transaction is initiated only if the first one is committed or aborted
- ◆ Database remains in a consistent state throughout the execution since transactions are executed one at a time
- ◆ Consider two transactions T1 and T2 in Fig 4.2.3. In a serial schedule either execute all the operations of T1 transaction and start T2, or execute all the

T1	T2
$r_1(A)$ $w_1(A)$	$r_2(B)$ $w_2(B)$

T1	T2
$r_1(A)$ $w_1(A)$	$r_2(B)$ $w_2(B)$

Fig. 4.2.3 Schedule 1-a serial schedule of T1 and T2

operations of T2 transaction and then start T1

4.2.2.2 Non-Serial Schedule

- ◆ Operations of multiple transactions are interleaved as shown in Fig 4.2.4
- ◆ Improve system throughput and resource utilization

T1	T2	T1	T2
$r_1(A)$		$r_1(A)$	
$w_1(A)$			$r_2(A)$
	$r_2(A)$	$w_1(A)$	
	$w_2(A)$	$r_1(B)$	
$r_1(B)$			$w_2(A)$
$w_1(B)$		$w_1(B)$	

Fig. 4.2.4 Schedule 2 -a non-serial schedule of T1 and T2

4.2.2.3 Serializable Schedule

- ◆ If the sequence of operations of a non-serial schedule is equivalent to a serial schedule, then the schedule is serializable
- ◆ Consider the following non-serial schedule of two transactions T1 and T2 in Fig 4.2.5

T1	T2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
$r_1(B)$	
$w_1(B)$	

Fig. 4.2.5 Schedule 3 - showing only the read and write instructions

- ◆ A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially. The above schedule is converted to a serial schedule as shown in Fig 4.2.6

T1	T2	T1	T2
$r_1(A)$			$r_2(A)$
$w_1(A)$		$r_1(A)$	$w_2(A)$
$r_1(B)$		$w_1(A)$	
$w_1(B)$	$r_2(A)$	$r_1(B)$	
	$w_2(A)$	$w_1(B)$	

Fig. 4.2.6 Schedule 4 - a Non serial schedule that is equivalent to Serial schedule

- ◆ Non-serial schedules may lead to data inconsistency. Hence it is necessary to identify whether non-serial schedules are serializable.
- ◆ Serializable schedule improves throughput and resource utilization

4.2.3 Types of Serializability

There are different methods to determine if a schedule is serializable. The most common ones include

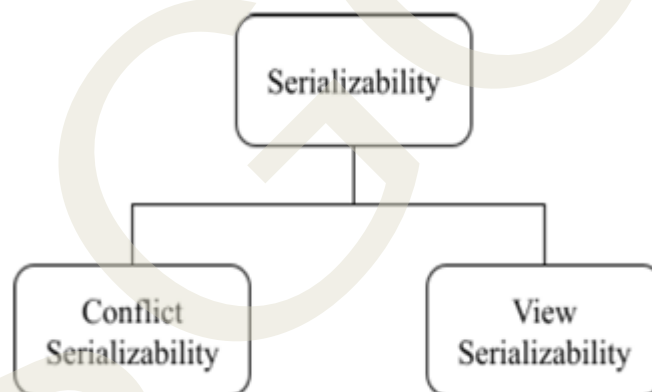


Fig. 4.2.7 Types of Serializability

4.2.3.1 Conflict Serializability

- ◆ A **conflict serializable schedule** allows transactions to run concurrently, but the outcome will be as if they had been executed sequentially, without overlapping.
- ◆ Two operations are said to be in **conflict** if they meet all the following conditions:
 - ◆ They belong to different transactions.
 - ◆ They operate on the same data item.

- ◆ At least one of the operations is a write

Consider a schedule in Figure 4.2.8

T1	T2
$r_1(A)$	$r_2(A)$
$w_1(A)$	
$r_1(B)$	

Fig. 4.2.8 Schedule 5-A conflicting schedule

- ◆ In the above schedule $w_1(A)$ and $r_2(A)$ are conflicting operations. They belong to different transactions. They operate on the same data and one of them is a write operation.
- ◆ To determine whether a schedule is conflict serializable, we can use a **precedence graph** (also called a **serializability graph**)

Algorithm for testing Conflict Serializability

Step 1: Create nodes for transactions

For each transaction T_i in the schedule S , create a node labeled T_i in the precedence graph.

Step 2: Handle read-write conflicts

For every pair of operations where T_j executes a read(A) after T_i executes a write(A), create a directed edge from T_i to T_j (i.e., $T_i \rightarrow T_j$).

Step 3: Handle write-read conflicts

For every pair of operations where T_j executes a write(A) after T_i executes a read(A), create a directed edge from T_i to T_j (i.e., $T_i \rightarrow T_j$).

Step 4: Handle write-write conflicts

For every pair of operations where T_j executes a write(A) after T_i executes a write(A), create a directed edge from T_i to T_j (i.e., $T_i \rightarrow T_j$).

Step 5: Check for cycles

The schedule S is conflict serializable if and only if the precedence graph has no cycles. If the graph contains a cycle, the schedule is not conflict serializable

Example: Consider the serial schedule in Figure 4.2.9

T1	T2
$r_1(A)$ $w_1(A)$	$r_2(A)$ $w_2(A)$

Fig. 4.2.9 Schedule 6-serial schedule

The precedence graph for the above schedule is as shown in Figure 4.2.10



Fig 4.2.10 Precedence graph of Schedule 6

Consider a non-serial schedule in Figure 4.2.11

T1	T2
$r_1(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$	$r_2(A)$ $w_2(A)$

Fig. 4.2.11 Schedule 7-non serial schedule

In Schedule 7 $w_1(A)$ and $r_2(A)$ are conflicting operations. The precedence graph for the above schedule is



Fig. 4.2.12 Precedence graph of Schedule 7 is equivalent to Schedule 6

The above graph is the same as the schedule explained in the previous example, hence the schedule is serializable.

4.2.3.2 View Serializability

- ◆ View serializability check if the outcome of the transactions is the same as it would be in a serial execution
- ◆ The schedule is view serializable if the read and write operations are arranged in such a way that the final database state is the same as if the transactions were executed serially, even if the operations are interleaved
- ◆ Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions
 1. Initial Read: If transaction T1 in schedule S1 reads a data item A for the first time, then in schedule S2, T1 must also read A as its first read. This ensures that the first read of a data item is the same in both schedules.
Example: If T1 reads A first in S1, then in S2, T1 should also read A first.
 2. Updated Read: If T1 reads A and T2 writes A in both schedules, then T1 must read the value that T2 wrote in both schedules.
Example: If T2 writes A in S1 and T1 reads A, then in S2, T1 should read the same value of A written by T2.
 3. Final Write: If T1 writes A in schedule S1 and T2 writes A in schedule S2, the final write of A in both schedules must be the same.
Example: If T1 writes A in S1 and T2 writes A in S2, both schedules should have the same final value of A.

Recap

- ◆ Transaction & Schedule – Understanding how multiple transactions execute in a database system.
- ◆ Operations in a Schedule – Read, write, commit, and abort operations within transactions.
- ◆ Types of Schedules – Serial schedule, non-serial schedule, and their differences.
- ◆ Serializable Schedules – Ensuring non-serial schedules produce the same result as a serial schedule.
- ◆ Conflict Serializability – Checking if a schedule maintains consistency using precedence graphs.
- ◆ View Serializability – Determining if a schedule is equivalent to a serial schedule based on read and write operations.
- ◆ Precedence Graph – A method to test conflict serializability by detecting cycles.
- ◆ Conflict Operations – Conditions when two operations conflict (different transactions, same data item, at least one write).

Objective Type Questions

1. What is the sequence of operations performed by multiple transactions in a database system called?
2. What type of schedule executes transactions one after the other without interleaving?
3. Which type of schedule allows interleaved execution of transactions?
4. What ensures that a non-serial schedule produces the same result as a serial schedule?
5. What type of serializability is determined using precedence graphs?
6. What is the method used to check conflict serializability?
7. What condition makes two operations conflicting?
8. What type of serializability is based on ensuring the same read and write order as a serial execution?
9. What is the final operation of a successful transaction?

10. What operation reverses a transaction's changes?
11. What term refers to the order of read, write, commit, and abort operations in a transaction?
12. What ensures database integrity when multiple transactions execute simultaneously?
13. What type of serializability considers initial read, updated read, and final write conditions?
14. What does a cycle in a precedence graph indicate about the schedule?
15. What type of transaction execution improves system throughput and resource utilization?

Answers to Objective Type Questions

1. Schedule
2. Serial
3. Non-serial
4. Serializability
5. Conflict
6. Precedence graph
7. Write
8. View
9. Commit
10. Abort
11. Schedule
12. Serializability
13. View
14. Non-serializable
15. Non-serial

Assignments

1. Explain the concept of serializability in database transactions. Why is it important?
2. Differentiate between conflict serializability and view serializability. Provide examples for each.
3. What is a precedence graph? How is it used to test conflict serializability?
4. Given a schedule of transactions, explain how you would determine whether the schedule is conflict serializable or not.
5. Write a short note on the conditions for a schedule to be view serializable. How does view serializability differ from conflict serializability?

Suggested Reading

1. Date, C. J., Kannan, A., & Swamynathan, S. (2006). *An introduction to database systems*. Pearson Education.
2. Elmasri, R., & Navathe, S. B. (2007). *Fundamentals of database systems* (5th ed.). Pearson.
3. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). Tata McGraw-Hill.
4. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). *Database system concepts* (6th ed.). Tata McGraw-Hill.

Reference

1. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Pearson Education.
2. Gray, J., & Reuter, A. (1992). *Transaction processing: Concepts and techniques*. Morgan Kaufmann.

Unit 3

Recovery

Learning Outcomes

The learner will be able to:

- ◆ define different types of failures in a database system
- ◆ list common failure types such as system crash, transaction failure and disk failure
- ◆ explain how database recovery helps maintain consistency and integrity
- ◆ describe the role of logs and checkpoints in database recovery
- ◆ summarize the steps involved in the recovery process in transaction management

Prerequisites

To understand Recovery, Types of Failure, Recovery Outline in Transaction Management (DBMS), it is important to have a fundamental knowledge of database management systems (DBMS) and how they operate. A key prerequisite is understanding transactions, which are sequences of operations performed as a single unit of work. For example, in an online banking system, transferring money from one account to another involves deducting the amount from one account and adding it to another. If a failure occurs in the middle of this process, the transaction must either be fully completed or fully undone to maintain consistency. This leads to the importance of ACID properties (Atomicity, Consistency, Isolation, and Durability), which ensure that transactions execute reliably even in the presence of failures.



Additionally, knowledge of database storage and file management is essential to understanding how data is physically stored and how failures impact data integrity. For instance, if a power outage or disk failure occurs while updating customer records in an e-commerce database, recovery mechanisms like logging and checkpointing help restore the system to a consistent state. Logs record all changes made during transactions, while checkpoints periodically save database states to minimize recovery time. Understanding these concepts helps in grasping how DBMS handles failures and ensures data is not lost or corrupted during unexpected disruptions. Additionally, knowledge of database storage and file management is essential to understanding how data is physically stored and how failures impact data integrity. For instance, if a power outage or disk failure occurs while updating customer records in an e-commerce database, recovery mechanisms like logging and checkpointing help restore the system to a consistent state. Logs record all changes made during transactions, while checkpoints periodically save database states to minimize recovery time. Understanding these concepts helps in grasping how DBMS handles failures and ensures data is not lost or corrupted during unexpected disruptions.

Keywords

Hardware Failures, Transaction Failures, Recovery Mechanisms, Redundancy, Checkpointing, Rollback, Synchronous Recovery.

Discussion

4.3.1 Failures and Recovery

Failures are disruptions that prevent a computing system or process from working as it should. These disruptions can occur due to various reasons, such as hardware issues, software errors, or unexpected external events. Recovery refers to the steps taken to restore the system to its proper state after a failure happens. Understanding failures and planning for recovery is crucial to keep systems reliable and ensure they work consistently, even when problems arise.

4.3.1.1 What is a Failure?

A failure happens when a system, application, or device cannot perform its intended task. For example, a hardware failure might occur if a hard drive crashes, or a software failure might happen due to a bug in a program. Failures can also result from external causes, like a power outage or a network disruption. They are typically categorized into types such as hardware failures, software failures, or transaction failures in databases. Knowing what type of failure has occurred helps in deciding how to fix the problem and prevent it in the future.

4.3.1.2 Importance of Recovery in Computing Systems

Recovery is essential to keep systems running smoothly after a failure. Without a good recovery plan, even small issues can cause significant problems like data loss, long

downtimes, or financial setbacks. Recovery techniques, such as saving checkpoints or rolling back incomplete tasks, help bring the system back to a consistent and functional state. For instance, in databases, recovery ensures that all completed transactions are saved, while unfinished ones are safely reversed. Overall, recovery strategies are key to ensuring that systems remain reliable and users can trust them to work correctly.

4.3.2 Types of Failures

Failures in computing systems can happen for many reasons, and understanding the type of failure is essential for solving the problem effectively. Failures can stem from hardware issues, software bugs, transaction problems, communication breakdowns, or human mistakes. Each type of failure affects the system differently, and addressing them ensures the system stays reliable and efficient.

4.3.2.1 Hardware Failures

Hardware failures occur when physical components of a computer system stop working as expected. This could involve hard drives crashing, memory becoming corrupted, or power supplies failing. These issues can lead to data loss or system downtime. For example, a server might stop functioning if its hard drive fails. To prevent such problems, solutions like creating data backups, using redundant hardware, and monitoring system health are commonly used.

4.3.2.2 Software Failures

Software failures happen when programs or applications don't work as they should. Common causes include bugs in the code, software updates that cause conflicts, or a lack of sufficient resources like memory or processing power. For instance, an application might crash because of a coding error. Fixing software failures often requires debugging, applying updates, or reverting to an earlier, stable version. Regular testing and maintenance can also help prevent software failures.

4.3.2.3 Transaction Failures

Transaction failures are specific to systems that handle multiple operations in a sequence, such as databases. These failures occur when a transaction cannot be completed successfully, often due to insufficient resources, a system crash, or a violation of data rules. For example, in an online purchase, if money is debited but the order is not confirmed, the system is left in an inconsistent state. Recovery processes like rolling back or retrying the transaction ensure that the data remains accurate and consistent.

4.3.2.4 Communication Failures

Communication failures occur when there is a problem in transferring data between devices or systems. This might happen because of network outages, poor connectivity, or transmission errors. For example, if a video call drops due to a weak internet connection, it is a communication failure. Systems address these issues by using techniques like error detection, retries, and backup communication channels to ensure data is successfully delivered.

4.3.2.5 Human Errors and Failures

Human errors are mistakes made by people while using or managing systems. These could include accidental deletion of important files, incorrect system configurations, or misuse of tools. For instance, a user might accidentally delete key data from a database. To reduce the chances of human errors, organizations can provide proper training, implement automated systems, and set up strict access controls. Even though human mistakes cannot always be avoided, user-friendly designs and clear instructions can help minimize their impact.

4.3.3 Impact of Failures

Failures in computing systems can cause a range of challenges, from reduced system performance to data loss and downtime. These disruptions can affect user experience, lead to financial losses, and compromise system reliability. Understanding the impact of failures helps in designing systems that are better equipped to handle such challenges effectively.

4.3.3.1 Effects on System Performance

Failures can significantly affect how efficiently a system operates. For instance, if a server crashes, the system may slow down or stop responding entirely. This not only delays processes but can also affect other systems connected to it. For example, a web application may face reduced performance due to issues with its database server. To address this, strategies like load balancing and regular monitoring can help maintain consistent performance.

4.3.3.2 Data Loss and Corruption

One of the most serious impacts of failures is the loss or corruption of valuable data. A hardware failure, such as a damaged hard drive, can result in the permanent loss of files. Similarly, software issues like incomplete transactions can corrupt databases, making the information unusable. Losing critical data, such as customer records or financial transactions, can harm both users and organizations. Regular data backups and recovery plans are essential to reduce the risks associated with such failures.

4.3.3.3 Downtime and System Availability

Failures often cause downtime, which means systems are unavailable for use. This can disrupt business operations, reduce productivity, and lead to financial losses. For example, if an online shopping platform goes down during a sale, customers may leave and revenue will be lost. Downtime also affects user trust, as people expect systems to be reliable. To minimize downtime, organizations use strategies like fault-tolerant system designs and disaster recovery plans to ensure systems remain available even during unexpected issues.

4.3.4 Recovery Mechanisms

Recovery mechanisms play a vital role in maintaining the stability and reliability of computer systems after a failure. They are designed to restore the system to its

normal state, prevent data loss, and minimize downtime. These techniques ensure that disruptions don't compromise the integrity of the system and allow processes to resume smoothly and efficiently. Let's explore some common recovery mechanisms.

4.3.4.1 Rollback Mechanism

The rollback mechanism helps the system return to its last known consistent state after a failure. This is especially useful in cases where a process or transaction is incomplete, such as in a banking transaction interrupted by a power outage. Rollbacks work by reversing any changes made during the incomplete transaction using stored logs or backups. For instance, if a user tries to transfer money and the process fails, a rollback ensures that no partial transfer or incorrect data is recorded. This mechanism safeguards the system's data integrity.

4.3.4.2 Checkpointing

Checkpointing involves saving a snapshot of the system's state at specific intervals. If a failure occurs, the system can restart from the most recent checkpoint instead of starting over completely. This approach saves time and reduces the chances of data loss. For example, when performing a long-running task, such as a scientific simulation, checkpointing ensures that progress is saved periodically. If an interruption happens, the simulation can pick up from the last checkpoint rather than starting from scratch. This method is widely used in large-scale systems to increase efficiency and reliability.

4.3.4.3 Replication and Redundancy

Replication and redundancy involve creating backup copies of data or system components to ensure smooth operation in case of a failure. For instance, cloud storage services maintain multiple copies of your data across different servers. If one server goes down, the data can still be accessed from another location. Similarly, redundant hardware components like power supplies or storage drives can take over when one fails, ensuring that the system remains operational. This approach is especially important in critical systems where continuous availability is essential, such as e-commerce or healthcare systems.

4.3.4.4 Deferred and Immediate Recovery

Deferred and immediate recovery strategies focus on when the recovery process begins after a failure. Deferred recovery waits until the system stabilizes or additional information is available before taking action. For example, in database systems, incomplete transactions can be resolved after the system has been restored. On the other hand, immediate recovery begins corrective actions as soon as a failure is detected. This is crucial in systems like online trading platforms, where even brief downtime can have significant consequences. Both approaches are designed to bring the system back to normal while considering the urgency and nature of the failure.

4.3.5 Phases in Recovery Process

Recovering from a system failure involves a series of steps that help restore stability and functionality. These steps focus on identifying the problem, fixing any

inconsistencies, and bringing the system back to normal operations. Each stage in the recovery process is essential to ensure data integrity and minimize the disruption caused by the failure. Below is a simplified explanation of these steps.

4.3.5.1 Detecting Failures

The first step is recognizing that a failure has occurred. Systems use tools like error logs, monitoring software, or alerts to identify problems as soon as they happen. For example, if a server goes offline, an automated alert can notify system administrators immediately. Detecting failures quickly helps to address issues before they grow into more significant problems, ensuring smoother recovery.

4.3.5.2 Restoring Data to a Consistent State

After identifying the failure, the next task is to make sure the system's data is consistent and reliable. This might involve rolling back incomplete transactions or using a previous checkpoint to recover lost data. For example, in an online shopping platform, if an order transaction is interrupted, rollback mechanisms ensure the customer's account balance and order details remain accurate. This step prevents data corruption and ensures the system remains trustworthy.

4.3.5.3 System Restart and Recovery Phases

The final step is restarting the system and gradually resuming normal operations. This might involve restarting critical services first, followed by non-essential ones. For instance, in a hospital's database system, restoring patient records and scheduling systems would take priority over less urgent functions. Restarting in phases ensures that the most important tasks are addressed first, reducing downtime for critical services.

These steps form the foundation of an effective recovery process. By detecting issues early, restoring data properly, and carefully restarting operations, organizations can minimize disruptions and ensure their systems run reliably.

4.3.6 Recovery Strategies

When a system experiences a failure, recovery strategies are put in place to bring it back to a working state. These strategies help minimize the impact of failures, ensuring the system gets back online quickly and with as little data loss as possible. Different strategies are used depending on the type of failure and how critical it is to restore the system.

4.3.6.1 Synchronous vs. Asynchronous Recovery

There are two main types of recovery: synchronous and asynchronous. In synchronous recovery, the system works to restore itself immediately when a failure occurs. It's like the system takes action right away, ensuring everything is in sync in real time. This is useful for systems where data consistency is very important, like in financial systems.

Asynchronous recovery, on the other hand, happens at a later time. The system can continue working normally, and the recovery happens in the background. This method doesn't interrupt the system's operations but may take longer to restore everything fully. Depending on the system's needs, one method might be preferred over the other.

4.3.6.2 Crash Recovery

Crash recovery is used to bring the system back to its last known working state after a sudden crash, like a power outage or system shutdown. The system looks at logs or saved points in time to figure out where it left off and restores everything back to a consistent state. For example, if you're in the middle of making a transaction on an online store and the system crashes, crash recovery ensures that the transaction is either completed or canceled, keeping everything in order.

4.3.6.3 Partial Recovery

Partial recovery focuses on fixing only the parts of the system that were affected by the failure, rather than restoring everything. For example, if a part of the system, like the database, crashes but other parts, like the user interface, are still working fine, partial recovery can restore just the database. This approach helps get the system back up and running quickly without having to fix things that aren't broken.

4.3.6.4 Full Recovery

In full recovery, the entire system is restored to its state before the failure. This is useful when a significant part of the system, like a server or database, crashes. Full recovery uses backups, logs, or saved states to ensure everything is returned to normal, from the data to the services. It might take longer but is crucial when the failure is severe and affects multiple components of the system.

Each of these strategies has its place, and the best one depends on how severe the failure is and how much downtime the system can afford. The goal is always to get the system back online as quickly and smoothly as possible while minimizing any potential data loss.

4.3.7 Significance of Backup and Recovery Plans

Backup and recovery plans are very important because they help ensure that data and systems can be restored when something goes wrong. These plans act like a safety net for organizations, allowing them to recover from problems such as hardware failure, software issues, or even natural disasters. Without proper backup and recovery plans, there's a higher chance of losing important data, which can lead to a lot of problems, including costly downtime and damage to the business.

Having a backup and recovery plan is a way to make sure business operations continue smoothly, even when things don't go as planned. These plans outline what needs to be done to restore data and systems, minimizing the effect of failures. With a good backup plan, organizations can quickly recover and keep their operations running, even in tough situations.

4.3.7.1 Regular Backup Techniques

Regular backups are one of the best ways to protect important data. By backing up data frequently, organizations can make sure that even if something goes wrong, they won't lose too much. There are different ways to back up data, such as full backups, incremental backups, and differential backups.

A full backup copies all of the data, while an incremental backup only saves the changes that have been made since the last backup. A differential backup saves the changes made since the last full backup. These different backup techniques make it easier to restore systems to various points in time, which can be very helpful when recovering from a problem.

4.3.7.2 Disaster Recovery Planning

Disaster recovery planning is about preparing for unexpected events that could disrupt business, like fires, floods, or cyberattacks. This plan helps organizations quickly recover and get back to work after a disaster, reducing the amount of time they are offline and preventing serious problems.

A disaster recovery plan includes steps for identifying which systems are most important, setting priorities for recovery, and defining how long it will take to restore systems. It also includes instructions for communicating during a crisis. Having a clear disaster recovery plan helps organizations react quickly to emergencies, allowing them to return to normal operations faster.

4.3.7.3 Best Practices in Data Protection

Protecting data is key to keeping it safe and secure. To make sure data is well-protected, organizations should follow some best practices. For example, they can encrypt backups to keep them secure, test backups regularly to make sure they'll work when needed, and store backup copies in different locations.

Other important practices include automating backups to avoid mistakes, restricting access to backups to prevent unauthorized users from making changes, and regularly reviewing backup procedures to ensure they are still effective. By following these best practices, organizations can be confident that their data is safe and can be recovered if something goes wrong.

Recap

- ◆ A failure refers to any event or condition that disrupts the normal operation of the database system, leading to incorrect behavior, loss of data integrity, or unavailability of services.
- ◆ Types of failures:
 1. Hardware Failures are malfunctions or breakdowns of physical components in a system.
 2. Software Failures are issues arising from defects or limitations within software applications or systems.
 3. Transaction Failures are the problems occurring during the execution of database transactions, leading to inconsistencies.
 4. Communication Failures are the breakdowns in data transmission between system components or networks.
 5. Human Errors are mistakes made by users or operators leading to system issues.
- ◆ Failures within Database Management Systems (DBMS) can have significant and multifaceted impacts on organizations, affecting operational efficiency, financial stability, and customer trust. Key consequences are system performance, data loss and corruption, downtime and system availability.
- ◆ Recovery mechanisms are essential strategies and processes designed to restore a database to a consistent state following various types of failures, such as system crashes, hardware malfunctions, or human errors.
- ◆ Rollback Recovery involves undoing the changes made by a transaction that has failed, restoring the database to its state before the transaction began.
- ◆ Checkpointing involves periodically saving the current state of the database and its logs to stable storage, reducing recovery time by limiting the amount of log data that must be processed after a failure.
- ◆ Redundancy and Replication implementing duplicate systems or data copies across different locations to ensure data availability and reliability, allowing for failover in case of system or media failures.
- ◆ Deferred and immediate recovery is when a transaction's updates are not applied to the database immediately. Instead, changes are first recorded in a log and held in the transaction's local workspace.
- ◆ Developing a robust database recovery strategy is essential for ensuring data integrity, availability, and system resilience in the face of various failures, including hardware malfunctions, software errors, and human mistakes.

- ◆ Regular data backups are essential for safeguarding information against unexpected events that can lead to data loss, such as hardware failures, cyberattacks, or human errors.
- ◆ Full Backups create complete copies of the entire database, providing a comprehensive recovery point but requiring significant storage space and time.

Objective Type Questions

1. Define a system crash in the context of database management systems.
2. What is the procedure for creating backup copies of data to guard against loss?
3. Which type of backup is the fastest to restore?
4. What phrase describes unforeseen circumstances that lead to system breakdowns, such as cyberattacks or natural disasters?
5. What strategy aids in system restoration following a disaster?
6. Which failure is defined as the problems occurring during the execution of database transactions, leading to inconsistencies?
7. Which process involves creating backup copies of data or system components to ensure smooth operation in case of failures?
8. In recovery, the system works to restore itself immediately when a failure occurs.
9. What is the primary purpose of a checkpoint in recovery mechanisms?
10. is an operation that undoes changes made by a transaction, reverting the database to its previous consistent state.

Answers to Objective Type Questions

1. Failure
2. Backup
3. Full
4. Disasters
5. Recovery

6. Transaction Failures
7. Replication and Redundancy
8. Synchronous
9. Suspend running transactions
10. Rollback

Assignments

1. Explain the different types of backup techniques (full, incremental, differential) and analyze the advantages and disadvantages of each in the context of data recovery.
2. Discuss the importance of having a disaster recovery plan in place for an organization. How do regular backups contribute to minimizing downtime during a system failure?
3. Compare and contrast synchronous and asynchronous recovery methods. Which method would be more suitable for a large enterprise with critical data, and why?
4. Evaluate the best practices in data protection and discuss how these practices can be integrated into a comprehensive backup and recovery plan to ensure business continuity.
5. Analyze the role of encryption in backup and recovery processes. How does encryption help in securing backup data, and what are the challenges associated with its implementation?

Suggested Reading

1. Beaulieu, A. (2009). *Learning SQL: Master SQL fundamentals*. O'Reilly Media.
2. Celko, J. (2005). *Joe Celko's SQL for smarties: Advanced SQL programming* (3rd ed.). Morgan Kaufmann.
3. Groff, J. R., Weinberg, P. N., & Oppel, A. J. (2002). *SQL: The complete reference* (Vol. 2). McGraw-Hill/Osborne.
4. Teate, R. M. P. (2021). *SQL for data scientists: A beginner's guide for building datasets for analysis*. John Wiley & Sons.

Reference

1. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Pearson Education.
2. Gray, J., & Reuter, A. (1992). *Transaction processing: Concepts and techniques*. Morgan Kaufmann.

SGOU

Unit 4

Deadlocks

Learning Outcomes

The learner will be able to:

- ◆ define deadlock and recognize how it can occur when processes have mutually exclusive access to multiple resources
- ◆ identify scenarios where deadlocks can paralyze process execution due to improper resource allocation and synchronization
- ◆ understand approaches to detect deadlocks after they occur and methods to recover from them
- ◆ define the different deadlock handling strategies in terms of system overhead and complexity

Prerequisites

You are already familiar with how operating systems manage multiple processes and threads to ensure smooth and efficient computing. This foundational knowledge sets the stage for delving deeper into critical concepts such as deadlocks, the conditions that lead to them, and strategies for their prevention.

Understanding process synchronization is essential, as it addresses the coordination of concurrent processes to prevent conflicts when accessing shared resources. This ensures that processes operate without interfering with each other, maintaining data integrity and system stability. Additionally, exploring multithreading and concurrency will enhance your comprehension of how multiple threads execute simultaneously, sharing resources and improving application performance. However, it's crucial to manage this concurrency carefully to avoid issues like deadlocks, where processes become stuck waiting for each other to release resources.

By connecting these concepts with your existing knowledge, you will gain a comprehensive understanding of how operating systems handle process synchronization and resource management, leading to more efficient and reliable computing systems.



Keywords

Deadlock Detection, Mutual Exclusion, Preemption, Wait-Die, Starvation, Deadlock Prevention.

Discussion

A deadlock occurs when two or more processes in a system are blocked forever, each waiting for a resource held by another. This results in a cycle of dependencies where no process can proceed, and the system is in a state of constant waiting. Deadlock can arise in concurrent systems, such as operating systems or databases, and needs to be managed carefully through detection, prevention, or avoidance techniques to ensure smooth execution of transactions.

4.4.1 Deadlock

An undesirable situation in which two or more transactions wait indefinitely for each other to release locks.

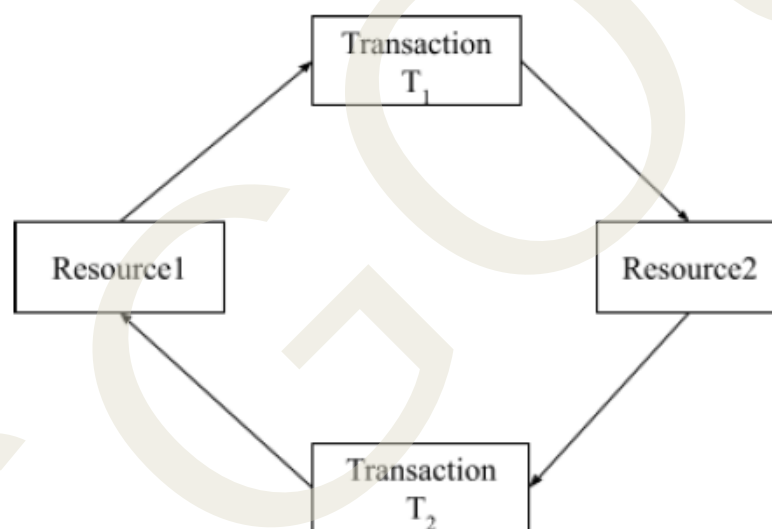


Fig. 4.4.1 A system in deadlock

In the above figure 4.4.1, T1 has Resource1 and needs Resource2. Similarly T2 has Resource2 and needs Resource1. Each of these processes needs other's resources to complete but neither of them is willing to give away their resources. So, T1 and T2 are in deadlock. All operations stop and stop forever unless the DBMS detects the deadlock and aborts one of the transactions.

4.4.2 Conditions for deadlock

A deadlock will occur if the following conditions hold:

a. Mutual Exclusion

There should be a resource that can only be owned by one transaction at a time as shown in fig 4.4.2.

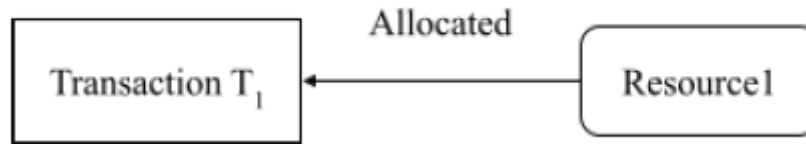


Fig. 4.4.2 Mutual exclusion condition

b. Hold and Wait

A transaction holds more than one resource and still requests for more resources as shown in fig 4.4.3.

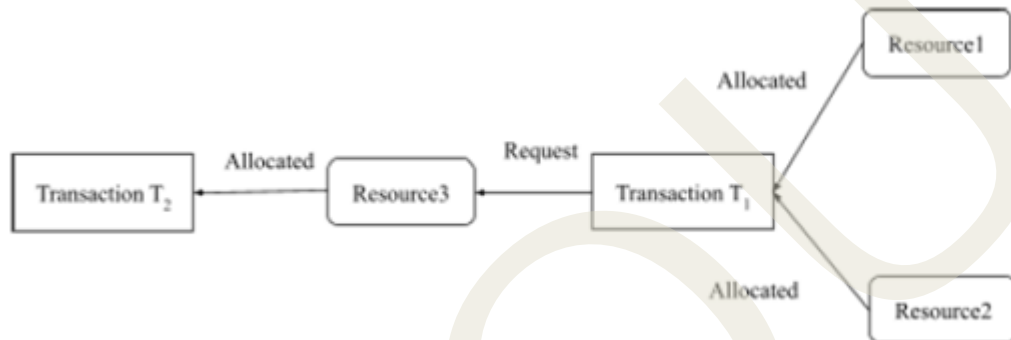


Fig. 4.4.3 Hold and Wait condition

c. No Preemption

Once a resource is allocated to a transaction, it cannot be forcibly taken away (preempted) by the system. A transaction can only release the resource voluntarily after completing its operation.

d. Circular Wait

A transaction is waiting for the resource held by the second transaction, which is waiting for the resource held by a third transaction and so on. The last transaction is waiting for a resource held by the first transaction. This forms a circular chain as shown in fig 4.4.4.

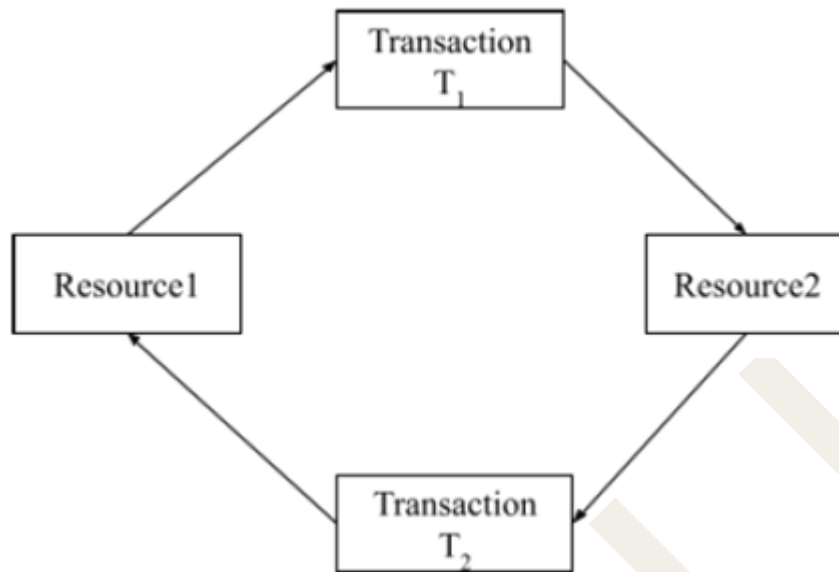


Fig. 4.4.4 Circular wait Condition

These conditions are called Coffman Conditions.

4.4.3 Deadlock Prevention

Deadlock prevention ensures that the system does not enter a deadlock state by imposing restrictions on transaction execution. It is particularly suitable for large databases where deadlock detection and resolution may be costly. The system closely examines each transaction before it is executed to ensure that it does not lead to a deadlock. If there is even a chance that a transaction leads to a deadlock, it should never be allowed to run.

A lock is a control mechanism associated with a database item that manages access to it. Before a transaction can read or modify a data item, it must acquire a lock; if another transaction holds the lock, it must wait.

4.4.3.1 Deadlock Prevention Protocols

1. Preemptive Locking

A transaction must lock all required data items in advance before execution. If any item cannot be locked, the transaction does not acquire any locks and must wait or restart. This method reduces deadlocks but limits concurrent execution.

How does Preemptive Locking Works?

a. Acquire All Locks in Advance

A transaction must ask for all the locks it needs before it starts doing any work. If any lock is unavailable, the transaction cannot start and must wait until it can lock all the resources it needs.

b. Waiting for Lock Availability

If some locks are already taken by other transactions, the requesting transaction must wait until those locks are released. During this waiting time, the transaction cannot do anything. It will only continue once it gets all the locks it needs.

c. No Partial Locking

A transaction cannot start using the data it has locked partially. It must wait until it locks all the required resources. If it still can't lock everything, it may restart and try to acquire the locks again.

d. Transaction Abort or Restart

If a transaction cannot get all the locks because other transactions are holding them, it may be aborted. The transaction can restart later, going through the process of acquiring all locks again before proceeding.

2. Ordering of Resources (Lock Ordering)

All data items in the database are assigned a predefined order, and transactions must request locks in that order. This prevents circular wait conditions. However, this method requires that the system or programmer know the order of items in advance, which may not always be practical.

For example: If resources are ordered as A, B, and C, a transaction must always request locks in this order (i.e., A first, then B, then C).

If a transaction requests a resource out of order, it is forced to wait or restart. This approach avoids situations where transactions form a circular wait.

4.4.3.2 Transaction Timestamp

A timestamp (TS) is a unique identifier assigned to each transaction, usually based on its start time. It helps determine the relative order of transactions in concurrency control mechanisms.

Notation: $TS(T)$ represents the timestamp of transaction T.

For example : If T_1 starts before T_2 , then T1 has an older timestamp:

$$TS(T_1) < TS(T_2).$$

There are two deadlock prevention schemes based on timestamps.

- ◆ Wait-Die Scheme (Non-preemptive)
- ◆ Wound-Wait Scheme (Preemptive)

A. Wait-Die Scheme (Non-preemptive)

Older transactions (with smaller timestamps) are allowed to wait for a younger transaction to release the resource. Younger transactions (with larger timestamps) die (are aborted) if they request a resource held by an older transaction.

If $TS(T_1) < TS(T_2)$, then T_1 is older than T_2 . T_1 is allowed to wait for the data item which will be free when T_2 has completed its execution. Otherwise abort T_1 and restart it later with the same timestamp.

Example:

- ♦ T_1 (older) requests a resource held by T_2 (younger) $\rightarrow T_1$ wait.
- ♦ T_2 (younger) requests a resource held by T_1 (older) $\rightarrow T_2$ is aborted ("dies") and restarted later.

B. Wound-Wait Scheme (Preemptive)

Older transactions "wound" (forcefully abort) younger transactions instead of waiting. If an older transaction requests a resource held by a younger transaction, it forces the younger transaction to abort (wound). If a younger transaction requests a resource held by an older transaction, it is forced to wait.

Example:

- ♦ If $TS(T_1) < TS(T_2)$, then T_1 is older than T_2 .
- ♦ T_1 (older) requests a resource held by T_2 (younger) $\rightarrow T_2$ is aborted ("wounded") and restarted later.
- ♦ T_2 (younger) requests a resource held by T_1 (older) $\rightarrow T_2$ wait.

4.4.4 Deadlock Detection

Deadlock detection is a more practical approach than prevention in systems where deadlocks are rare and preventing them entirely may reduce performance. The DBMS periodically checks whether a deadlock has occurred by examining waiting transactions and their resource dependencies. A common method for detecting deadlocks is using a Wait-for Graph (WFG).

4.4.4.1 Wait-for graph

A Wait-for Graph is drawn based on the currently executing transactions and locks on data items. In the Wait-for Graph each transaction is a node. A node is created for each current transaction as shown in fig 4.4.5.

If a transaction T_1 is waiting for a data item X that is locked by another transaction T_2 , a directed edge is drawn from T_1 to T_2 .

An edge from $T_1 \rightarrow T_2$ means T_1 is waiting for a resource held by T_2 . When T_2 releases the lock on X , the corresponding edge is removed from the graph. If the graph contains a cycle, it indicates a deadlock state. The system must check for a cycle every time an edge is added to the graph.

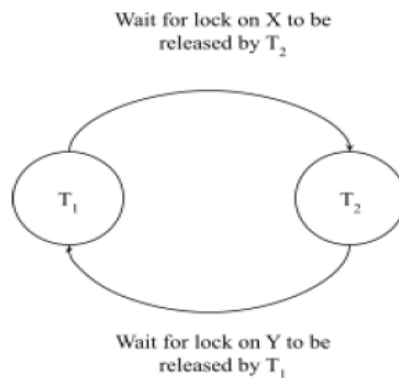


Fig. 4.4.5 Wait-for graph

4.4.4.2 Timeouts in Deadlock Handling

The system assigns a timeout period to each transaction. If a transaction waits longer than the timeout period while trying to acquire a lock, it is assumed to be in a deadlock. The system may abort the transaction to break the deadlock and free up resources. This approach is simple but may lead to unnecessary transaction rollbacks if the timeout is set too short.

4.4.5 Starvation

Starvation occurs when a transaction is indefinitely delayed due to an unfair lock allocation scheme. This happens when certain transactions are continuously given priority over others, preventing some transactions from ever acquiring the required resources. As a result, the affected transaction remains waiting indefinitely and cannot proceed.

Starvation can occur due to:

- ◆ Priority-based scheduling, where high-priority transactions are always executed first.
- ◆ Frequent rollbacks of low-priority transactions.
- ◆ Resource preemption policies that always favor certain transactions.

Recap

- ◆ Locks is a mechanism to control access to shared resources in concurrent programming and prevent conflicts by ensuring that only one process/thread accesses a resource at a time.
- ◆ Deadlock is a situation where a set of processes are blocked because each process holds a resource and waits for another resource held by another process, leading to a standstill where no process can proceed.
- ◆ **Conditions for Deadlock (Coffman Conditions):**
 - **Mutual Exclusion:** At least one resource is held in a non-shareable mode.
 - **Hold and Wait:** A process holds at least one resource and waits for additional resources held by other processes.
 - **No Preemption:** Resources cannot be forcibly taken from processes; they must release them voluntarily.
 - **Circular Wait:** A set of processes exists such that each process is waiting for a resource held by the next process in the set.
- ◆ **Deadlock Prevention Strategies:**
 - **Eliminate Mutual Exclusion:** Make resources shareable when possible.
 - **Avoid Hold and Wait:** Require processes to request all needed resources simultaneously or release all held resources before requesting new ones.
 - **Allow Preemption:** Forcefully take resources from processes when necessary.
 - **Prevent Circular Wait:** Establish a hierarchy for resource allocation to avoid circular chains of waiting processes.
- ◆ **A transaction timestamp** is a unique identifier assigned to a transaction at its initiation, indicating the precise time the transaction began.
- ◆ **Starvation** is as indefinite blocking, starvation occurs when a process is perpetually denied access to the resources it needs for execution because other processes are continually favored.
- ◆ **Recovery strategies** are essential for ensuring data integrity and availability in the event of system failures.

Objective Type Questions

1. Which condition is required for a deadlock to be possible?
2. Which state of the system can allocate resources to each process in some order and still avoid a deadlock?
3. How can the circular wait condition be prevented?
4. The variable that indicates the status of a data item in the database.
5. When should deadlock checks be performed in an operating system?
6. What term describes a situation where a process is perpetually denied necessary resources?
7. Which graph is used to visually determine the occurrence of deadlock?
8. What is the primary focus of deadlock recovery in operating systems?
9. What is a necessary condition for deadlock to occur in a system?
10. What role does preemption play in preventing deadlock?
11. A cycle in the Wait-for graph indicates a deadlock. True or False.
12. Wound-wait is a non preemptive technique. True or False
13. In which technique older transactions have to wait for younger transactions to release the resource.
14. A unique identifier assigned for each transaction based on the order in which they are started.
15. A condition refers to a scenario where a process holds at least one resource and is simultaneously waiting to acquire additional resources that are currently held by other processes.

Answers to Objective Type Questions

1. Mutual exclusion, hold and wait, no preemption, and circular wait.
2. Safe State.
3. Preventing Circular Wait.
4. Lock.
5. Deadlock Checks Timing
6. Starvation.

7. Wait-for graph.
8. Terminate deadlocked processes and reclaim resources.
9. Mutual exclusion.
10. Preemption.
11. True.
12. False.
13. Wait-die.
14. Timestamp.
15. Hold and wait

Assignments

1. Explain the deadlock with an example. Describe why deadlock is a critical issue in concurrency control
2. Discuss the four Coffman conditions necessary for deadlock. Provide a real-life analogy for each condition.
3. Describe the different deadlock prevention techniques. How do these techniques help in avoiding deadlock?
4. What is a wait-for graph? Explain how it is used for deadlock detection with an example.
5. Compare and contrast the wait-die and wound-wait schemes for deadlock prevention. Which one is more efficient and why?

Suggested Reading

1. Date, C. J., Kannan, A., & Swamynathan, S. (2006). *An introduction to database systems*. Pearson Education.
2. Elmasri, R., & Navathe, S. B. (2007). *Fundamentals of database systems* (5th ed.). Pearson.
3. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). Tata McGraw-Hill.
4. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). *Database system concepts* (6th ed.). Tata McGraw-Hill.

Reference

1. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Pearson Education.
2. Gray, J., & Reuter, A. (1992). *Transaction processing: Concepts and techniques*. Morgan Kaufmann.



NoSQL Databases

Unit 1

Introduction to NoSQL

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- ◆ define NoSQL and its purpose in database management
- ◆ identify key reasons for the emergence of NoSQL databases
- ◆ list the main characteristics of NoSQL databases
- ◆ list BASE properties
- ◆ recall the differences between NoSQL and RDBMS

Prerequisites

Before exploring NoSQL databases, it is essential to understand how traditional databases store and manage data. You might already be familiar with Relational Database Management Systems (RDBMS) such as MySQL, PostgreSQL, or Oracle, which structure data into tables with rows and columns. These databases operate based on predefined rules and use Structured Query Language (SQL) to efficiently store, retrieve, and manage information.

Now, consider modern digital platforms like social media (Facebook, Instagram, Twitter), e-commerce sites (Amazon, eBay), and messaging apps (WhatsApp, Telegram). These applications generate vast amounts of data every second in various formats, including text, images, videos, and user interactions. Traditional databases can struggle to process such large, dynamic data efficiently, leading to performance limitations and scalability issues.

This is where NoSQL databases become valuable. They provide a more flexible, scalable, and high-performance solution for handling big data, real-time applications, and distributed systems. In this lesson, we will examine the emergence of NoSQL, its key characteristics, and how it differs from RDBMS. By the end of this discussion, you will gain insights into how NoSQL databases enable businesses to manage complex and rapidly growing data effectively.



Keywords

NoSQL, BASE properties, CAP theorem, Flexibility, Big Data

Discussion

5.1.1 NoSQL

NoSQL, which stands for "Not Only SQL," is a type of database system designed to store and manage large amounts of data, particularly semi-structured and unstructured data (such as text, images, and videos). Unlike traditional databases that use fixed tables and strict rules, NoSQL databases are more flexible and can handle different types of data easily. They are also built for scalability, meaning they can grow as needed by adding more servers. This makes them perfect for modern applications that require fast and real-time data processing.

The demand for NoSQL databases has increased due to the vast amount of data generated by social media, smart devices (IoT), and real-time applications. Traditional databases often struggle to manage this data efficiently, making it harder for businesses to scale up. NoSQL databases solve this problem by using distributed systems (storing data across multiple servers) and allowing flexible data storage, making them faster and more efficient for large-scale applications.

An example of NoSQL in action is social media platforms like Instagram or Twitter. Millions of users post, like, and comment every second, creating huge amounts of data in different formats (text, images, videos, etc.). Traditional databases might struggle to handle this constantly changing data, but NoSQL databases like MongoDB and Cassandra provide a better solution. These databases allow for fast data processing, quick updates, and smooth scaling as the number of users grows. Due to their speed, scalability, and flexibility, NoSQL databases are widely utilised in large-scale applications that require processing massive amounts of data efficiently.

5.1.2 Emergence of NoSQL

As data volumes grew exponentially, relational databases struggled to scale efficiently while maintaining performance. The increasing need for high-speed data processing, flexible schemas, and distributed storage led to the development of NoSQL databases.

Key factors contributing to the emergence of NoSQL include

1. **Big Data Growth** : The explosion of unstructured and semi-structured data from social media, IoT devices, and e-commerce platforms.
2. **Scalability Challenges** : Traditional relational databases scale vertically (adding more power to a single server), which is costly and limited. NoSQL enables horizontal scaling (adding more servers).
3. **Cloud and Distributed Computing** : Companies needed databases that could efficiently run across multiple distributed servers.

4. **Flexible Data Models** : Unlike relational databases, NoSQL does not require a fixed schema, making it easier to adapt to evolving data requirements.
5. **Real-time Processing Needs** : Applications like social media, recommendation systems, and financial transactions demand fast reads and writes.

5.1.3 Characteristics of NoSQL Database

1. **Dynamic Schema**: NoSQL databases do not enforce a fixed schema, allowing data structures to evolve without requiring schema modifications or migrations. This flexibility makes them ideal for applications where data formats frequently change.

Example: A blogging platform where each blog post can have different metadata (e.g., some have images, others do not).

2. **Horizontal Scalability**: NoSQL databases are designed to scale horizontally by adding more servers (nodes) to a database cluster. This architecture efficiently manages increasing data volumes and high traffic loads.

Example: An online shopping website that adds more database servers as more users visit during a holiday sale.

3. **Document-Based Model**: Some NoSQL databases, such as MongoDB, store data in a semi-structured, schema-less format like JSON or BSON. This enables flexibility in storing complex and hierarchical data.

Example: An online bookstore where different books have varying attributes (some have authors, some have translators).

4. **Key-Value-Based Model**: Certain NoSQL databases, such as Redis, utilize a key-value storage system, where each data entry consists of a unique key and its corresponding value. This model is highly efficient for fast data retrieval.

Example: A gaming leaderboard where each player's score is stored using their username as a key.

5. **Column-Family-Based Model**: Some NoSQL databases, such as Cassandra, store data in a column-oriented format rather than traditional row-based storage. This design enhances performance for queries that involve reading large datasets.

Example: A telecom company storing customer call records, where each customer has multiple call logs.

6. **Distributed and High Availability**: NoSQL databases are inherently distributed across multiple nodes, ensuring high availability and automatic handling of node failures through data replication.

Example: a banking app that stays online even if one server fails.

7. **Flexibility:** NoSQL databases provide greater flexibility by allowing developers to store diverse data types without strict constraints. This adaptability is particularly useful for rapidly evolving applications.

Example: A messaging app where users can send text, images, videos, and voice notes in the same chat.

8. **High Performance:** NoSQL databases are optimized for fast read and write operations, making them well-suited for high-traffic, big data, and real-time applications.

Example: A stock market app that updates live prices instantly for millions of users.

5.1.4 BASE Properties in NoSQL

NoSQL databases adhere to the BASE properties rather than the strict ACID rules employed in traditional databases. BASE helps NoSQL databases handle large amounts of data quickly by prioritizing availability and speed over strict accuracy.

1. **Basically Available :** The system continues to operate even if some components experience issues.

Example: On an e-commerce platform like Amazon, even if a specific warehouse's database is slow, the website remains accessible, allowing users to still place orders.

2. **Soft State :** The data in the system may undergo changes over time, even without new updates, as different copies synchronize.

Example: When you update your profile picture on Facebook, it may take a few moments for all users to see the change since various servers update at different rates.

3. **Eventually Consistent :** While data may appear inconsistent for a short period, it will ultimately become accurate across all systems.

Example: After posting a tweet on Twitter, some users may see it immediately, while others might experience a delay. However, after some time, everyone will have access to the correct post.

5.1.5 CAP Theorem

The CAP Theorem states that a distributed database system can only guarantee two out of three properties at the same time, as shown in Fig.5.1.1. Since NoSQL databases are distributed (spread across multiple servers), they must balance and choose which features to focus on.

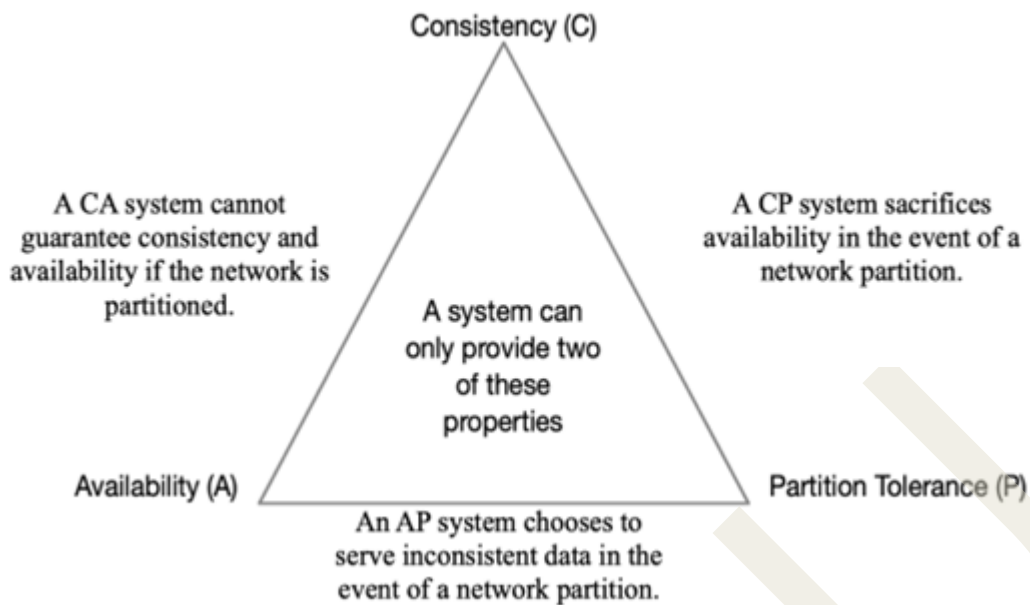


Fig. 5.1.1 CAP Theorem

1. **Consistency (C)** : Every time data is read, it always shows the latest, correct version across all servers.

Example: In a banking system, if a user transfers money, their updated balance should be immediately visible across all branches.

2. **Availability (A)** : The system guarantees a response to every request, even if some servers are down.

Example: In an online shopping website like Amazon, users should still be able to browse products and place orders, even if some database servers are temporarily unreachable.

3. **Partition Tolerance (P)** : The system continues to function even if communication between servers is disrupted due to network failures.

Example: In a messaging app like WhatsApp, messages should still be sent and received even if some network connections are unstable.

5.1.6 Comparison with RDBMS

NoSQL and RDBMS are two major types of databases, each suited for different use cases. RDBMS (Relational Database Management System) follows a structured approach with tables, enforcing strict consistency through ACID properties. In NoSQL, databases offer flexibility, scalability, and high performance, making them ideal for handling big data and real-time applications. Table 5.1.1 below compares their key differences.

Table 5.1.1 Comparison Table

RDBMS(Relational Database Management System)	NoSQL (Not Only SQL)
Vertically Scalable	Horizontally Scalable
Follows ACID properties	Follows BASE properties
Decreases performance when dealing with large amount of semi-structured and unstructured data(Big Data)	Performance, scalability and flexibility required for Big Data provided by the NoSQL databases
Limited Scalability	No limit on Scalability
Change Management is difficult due to rigid schema	Schema free and Change Management is easy
Handle limited data	Handle large volumes of data especially in Big Data
Suffers from single point of failure	Distributed nature provides availability to users all the time in the presence of hardware failures
Data Replication problem	Support automatic data replication

5.1.7 Relevance of NoSQL

NoSQL databases have become essential in today's data-driven world, especially for applications that need scalability, flexibility, and high performance. Businesses and technology firms widely adopt NoSQL databases for several reasons:

1. Handling Massive Data Volumes

- ◆ NoSQL databases efficiently manage huge amounts of data (terabytes or even petabytes).

Example: Financial institutions, healthcare systems, and e-commerce platforms store and analyze large datasets using NoSQL.

2. Support for Real-time Applications

- ◆ Many modern applications require fast data access and updates.

Example: Streaming services like Netflix and gaming platforms need low latency (fast response times) to provide smooth user experiences.

3. Scalability for Growing Businesses

- ◆ NoSQL databases can scale horizontally (by adding more servers), making them ideal for companies experiencing rapid growth.

Example: E-commerce platforms like Amazon scale easily to handle increasing customer traffic during peak seasons, such as Black Friday.

4. Microservices Architecture

- ◆ Modern applications use microservices, where different services run independently and require a flexible database.

Example: In a food delivery app, separate databases can handle orders, payments, and delivery tracking efficiently.

5. Cloud-native and Distributed Environments

- ◆ NoSQL databases integrate seamlessly with cloud computing services like AWS, Google Cloud, and Azure.

Example: Social media platforms like Instagram and Twitter store user-generated content across multiple cloud servers to ensure availability and speed.

Recap

- ◆ NoSQL stands for "Not Only SQL."
- ◆ It is designed to handle semi-structured and unstructured data efficiently.
- ◆ NoSQL databases emerged due to the rapid growth of big data, real-time applications, cloud computing, IoT, and social media.
- ◆ NoSQL databases are schema-less, horizontally scalable, highly flexible, high-performing, and distributed.
- ◆ It follows BASE properties – Basically Available, Soft State, Eventually Consistent.
- ◆ CAP Theorem states that a distributed system can ensure only two out of three properties: Consistency, Availability, and Partition Tolerance.
- ◆ These databases are highly relevant for handling large-scale data, high-speed processing, and scalable applications.
- ◆ NoSQL is widely used in applications such as social media platforms, real-time analytics, recommendation systems, and IoT.

Objective Type Questions

1. What does NoSQL stand for?
2. What types of data are NoSQL databases designed to handle?
3. How do NoSQL databases achieve horizontal scalability?
4. What are the BASE properties in NoSQL databases?
5. What are the three properties of the CAP theorem?
6. How do NoSQL databases ensure high availability and fault tolerance?

7. What property of NoSQL databases allows temporary inconsistencies?
8. How does a document-based NoSQL database store data?
9. Why are NoSQL databases highly available?
10. What is the primary purpose of key-value NoSQL databases?

Answers to Objective Type Questions

1. Not Only SQL
2. Semi-structured and unstructured data
3. By adding more servers (horizontal scaling)
4. Basically Available, Soft State, Eventually Consistent
5. Consistency, Availability, Partition Tolerance
6. By distributing data across multiple servers
7. Soft State
8. In JSON or BSON format
9. Due to distributed architecture and replication
10. Fast data retrieval using key-value pairs

Assignments

1. Explain the key differences between NoSQL and RDBMS databases.
2. Describe the BASE properties of NoSQL databases with examples.
3. Discuss the CAP theorem and its relevance to NoSQL databases.
4. Explain the different types of NoSQL databases with suitable examples.
5. Describe the role of NoSQL databases in handling big data and real-time applications.

Suggested Reading

1. Cattell, R. (2011). *Scalable SQL and NoSQL data stores*. ACM SIGMOD Record, 39(4), 12-27.
2. Han, J., Haihong, E., Le, G., & Du, J. (2011). *Survey on NoSQL database*. 2011 6th International Conference on Pervasive Computing and Applications, 363-366.
3. Sadalage, P. J., & Fowler, M. (2012). *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Addison-Wesley.
4. Stonebraker, M. (2012). *New opportunities and challenges in NoSQL databases*. Communications of the ACM, 55(4), 10-11.
5. Vaish, G. (2013). *NoSQL: Database for storage and retrieval of data in cloud*. McGraw Hill.

Reference

1. Grolinger, K., Higashino, W. A., Tiwari, A., & Capretz, M. A. M. (2013). *Data management in cloud environments: NoSQL and NewSQL data stores*.
2. Moniruzzaman, A. B. M., & Hossain, S. A. (2013). *NoSQL database: New era of databases for big data analytics—classification, characteristics, and comparison*.
3. Li, X., & Manoharan, S. (2013). *A performance comparison of SQL and NoSQL databases*.
4. Sharma, S., Dave, M., & Patel, A. (2021). *Comparative study of NoSQL databases for big data storage and processing*.
5. Strauch, C., Sites, R., & Manhart, P. (2011). *NoSQL databases: An overview and analysis*. Stuttgart Media University.

Unit 2

Types of NoSQL Databases

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- ◆ define NoSQL databases and list their four main types
- ◆ identify the key features of Key-Value Stores, Document-Oriented Databases, Column-Family Stores, and Graph-Based Databases
- ◆ list examples of each type of NoSQL database
- ◆ explain the storage structure of Column-Family Stores and how it differs from relational databases

Prerequisites

In our everyday digital lives, we interact with vast amounts of data. Think about social media platforms like Facebook or Instagram, where millions of users share posts, comments, and messages every second. Traditional databases store data in structured tables, but when dealing with such massive and diverse information, a more flexible and efficient way of handling data is needed.

This is where NoSQL databases come in. Unlike traditional databases, NoSQL databases allow for more dynamic data storage, making them ideal for handling large-scale applications like e-commerce websites, cloud storage, and recommendation systems.

To understand NoSQL databases better, you should have a basic idea of how traditional databases work, especially relational databases (SQL). You might have learned that SQL databases store data in tables with predefined structures. However, NoSQL databases take a different approach, offering more flexibility and scalability.

In this unit, we will explore different types of NoSQL databases, including Key-Value Stores, Document-Oriented Databases, Column-Family Stores, and Graph-Based Databases. Each type has unique features that make it suitable for specific applications. By the end of this discussion, you will understand how NoSQL databases function and why they are essential in handling modern data storage challenges.



Keywords

NoSQL databases, Key-Value Stores, Document-Oriented Databases, Column-Family Stores, Graph-Based Databases

Discussion

5.2.1 Introduction

NoSQL (Not Only SQL) databases are designed to handle large amounts of unstructured or semi-structured data. Unlike traditional relational databases (RDBMS), NoSQL databases do not use tables and schemas. They are more flexible and scalable, making them ideal for modern applications. NoSQL databases are classified into four main types based on their data models:

1. Key-Value Stores
2. Document-Oriented Databases
3. Column-Family Stores
4. Graph-Based Databases.

5.2.2 Key-Value Stores

A Key-Value Store is a type of NoSQL database that stores data as pairs of keys and values. Each key is unique and is used to retrieve its corresponding value quickly. The values can be basic types like numbers and text, or more complex data structures. A key-value store works similarly to a relational database but has only two parts: a key and its corresponding value. A key-value store is simpler. It has only two parts: a key and a value. The key is used to find and retrieve the stored value.

A key-value database saves information in a simple way, where each piece of data is linked to a unique key. Imagine you are creating a system for an online bookstore. You need to store details about each customer, like their ID, name, email, and address. Since all these details belong to one customer, you can assign a unique customer ID and use it to create keys for each piece of data.

For example, if the first customer has an ID of 1001, the but has only two parts of information, it can be stored like this:

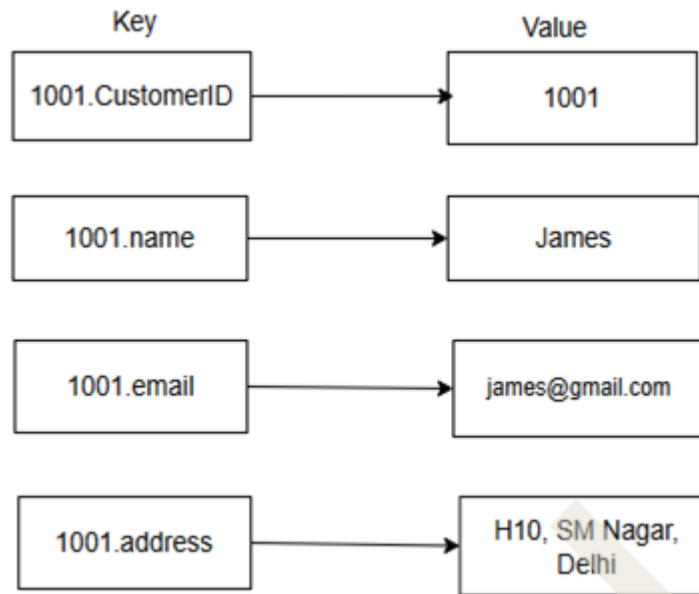


Fig. 5.2.1 Key-value store of an online bookstore

5.2.2.1 Features of Key-Value Store

1. Simple Data Storage

A key-value store organizes data in the form of a key and its associated value, similar to a dictionary or a hash table. Each key acts as a unique identifier for the value it stores. This simplicity in structure allows for fast implementation and easy management, especially in applications that do not require complex relationships or schemas.

2. Fast Access

Key-value stores are designed to fetch data quickly using unique keys. Since each piece of data can be found directly through its key, there is no need to search through all the data. This makes data access very fast. Such speed is helpful in situations like managing user sessions, storing temporary data (caching), and running real-time applications where quick responses are important.

3. Scalable

One of the key advantages of key-value databases is their ability to scale horizontally. This means they can manage large amounts of data by spreading it across several servers or nodes in a distributed system. As a result, they can maintain high performance even when the data volume grows. This feature makes them well-suited for large-scale web applications that require continuous availability and the ability to handle failures without losing data or service.

4. Flexible Data Types

Key-value databases are not limited to storing only simple data types like strings or numbers. They can also store complex data types such as JSON, XML, or even binary objects. This flexibility allows developers to store structured, semi-structured, or unstructured data without needing to alter the database schema.

5. No Fixed Structure

Unlike relational databases, key-value stores do not require a fixed schema or predefined structure. Each key-value pair is independent, meaning different keys can store different types of data. This schema-less nature enables rapid development and adaptation to changing data requirements without the need for database redesign.

5.2.2.2 Examples of Key-Value Store Databases

1. **Redis:** It is a popular in-memory key-value database, which means it stores data directly in the computer's memory (RAM) instead of on disk. This makes it extremely fast for reading and writing data. Redis is also used in real-time apps to track what users are doing or to keep scoreboards for games.
2. **Amazon DynamoDB** is a fully managed key-value and document database offered by Amazon Web Services (AWS). It is designed to handle large amounts of data and millions of requests per second, making it ideal for big applications like e-commerce websites, mobile apps, and online games.
3. **Memcached** is a high-speed caching system that stores frequently accessed data in memory to make websites and applications run faster. Instead of going to the main database every time, the system first checks Memcached to see if the needed data is already available. This reduces the load on the database and improves performance.

5.2.3 Document-Oriented Databases

A document-oriented database is a type of NoSQL database that stores data in a document-based format instead of using tables like traditional databases. These databases store data as self-contained, flexible documents, commonly in JSON (JavaScript Object Notation), BSON (Binary JSON), or XML (Extensible Markup Language). Unlike relational databases that enforce a strict schema, document databases allow for schema-less storage, making them ideal for applications requiring flexibility and scalability.

Consider an online shopping website where each product has details such as name, price, category, stock availability, and customer reviews. In a relational database, this data would be split across multiple tables, requiring complex joins to retrieve information. However, in a document-oriented database like MongoDB, all product-related data can be stored in a single document, making retrieval faster and more efficient.

Example document in MongoDB:

```
{  
  
  "productID": 101,  
  
  "name": "Wireless Headphones",  
  
  "price": 59.99,  
  
}
```

```
"category": "Electronics",  
"stock": 120,  
"reviews": [  
  {"customer": "Alice", "rating": 5, "comment": "Great sound quality!"},  
  {"customer": "Bob", "rating": 4, "comment": "Good battery life."}  
]  
}
```

5.2.3.1 Features of Document-Oriented Databases

1. Document-Based Storage

Document-oriented databases store data in the form of individual documents, typically using formats such as JSON, BSON, or XML. Unlike relational databases that rely on structured tables with predefined rows and columns, this model allows each document to contain all relevant data for a particular entity, thereby supporting more natural data modeling and easier data management.

2. Schema-Less Structure

These databases do not enforce a fixed schema. As a result, documents within the same collection can have different sets of fields and structures. This schema flexibility enables developers to modify or extend data models without the need for altering the entire database design, thereby supporting rapid application development and evolving data requirements.

3. Faster creation and maintenance

In document-oriented databases, documents can be created quickly because they do not require a fixed structure. This makes it easier to add new data without a complex setup. Once the documents are created, they usually need very little maintenance. As a result, developers can save time and focus more on building and improving the application rather than managing the database.

4. No foreign keys are needed

In document-oriented databases, each document is independent and usually contains all the necessary information within itself. Since documents are not strongly linked to one another through dynamic relationships, there is no need to use foreign keys as in relational databases. This independence simplifies data storage and reduces the complexity of managing relationships between data.

5.2.3.2 Examples of Document-Oriented Databases

1. MongoDB

MongoDB is one of the most commonly used document databases. It stores data in a flexible format similar to JSON, which allows easy changes to the structure of data. MongoDB supports searching, filtering, and organizing data efficiently, making it useful for many web and mobile applications.

2. CouchDB

CouchDB stores data in JSON format and is designed to be user-friendly. It supports copying and synchronizing data across different devices, which makes it suitable for applications that need to work smoothly in both online and offline environments.

3. Firebase Firestore

Firebase Firestore is a cloud-based document database designed for mobile and web applications. It allows data to be stored and accessed in real time, enabling apps to update information instantly across all devices.

4. Amazon DocumentDB

Amazon DocumentDB is a NoSQL document database service designed to handle large-scale applications. It offers high performance, scalability, and is compatible with MongoDB, making it suitable for enterprise-level cloud environments.

5.2.4 Column-Family Stores

A column-family database is a type of NoSQL database that stores data in a way that is different from traditional relational databases. Instead of organizing data in rows and tables like in SQL databases, it organizes data into columns that are grouped together into families.

Imagine you have a spreadsheet that stores student information. Each row represents a student, and each column represents specific details about them, as shown in Table 5.2.1

Table 5.2.1 Student Data

Student_ID	Name	Age	Email	Marks
101	Helen	22	helen@gmail.com	86
102	James	21	james@gmail.com	90
103	Alex	20	alex@gmail.com	77

In a relational (SQL) database, when you search for all students who scored more than 80 marks, the database must scan through each row to check the "Marks" column. This method is called row-based storage, and it works well for transactional applications.

In column-family database storage, instead of storing data row by row, a column-family database groups related columns together. Let's break the same data into column families:

Table 5.2.2 Column Family: Personal_Info

Student_ID	Name	Age	Email
101	Helen	22	helen@gmail.com
102	James	21	james@gmail.com
103	Alex	20	alex@gmail.com

Table 5.2.3 Column Family: Academic_Info

Student_ID	Marks
101	86
102	90
103	77

Now, if you want to find all students who scored more than 80 marks, the database only needs to look at the Academic_Info column family, instead of scanning every row. This makes the search faster and more efficient, especially for big data applications.

5.2.4.1 Features of Column- Family Stores

1. Column-Oriented Storage

Column-family databases store data by columns rather than by rows. This means that all the values of a particular column are stored together, which improves performance for read and write operations involving specific columns.

2. Column Families

Related columns are grouped into structures called column families. Each column family stores data that is usually accessed together. This organization helps in retrieving relevant data more efficiently and improves overall performance.

3. Scalability

Column-family stores are built to handle very large volumes of data. They are well suited for big data applications, as they can scale horizontally across many servers, ensuring that data storage and processing remain efficient even as data grows.

4. Fast Reads and Writes

Since data is stored by columns, column-family stores can quickly access specific pieces of information without scanning entire rows. This structure makes read and write operations faster, especially when working with large datasets and when only a few columns are needed from each record.

5.2.4.2 Examples of column-family store

1. Apache Cassandra

Apache Cassandra is a highly scalable and fault-tolerant column-family database designed to handle large volumes of data across multiple servers. It provides high

availability with no single point of failure and is used by major companies such as Facebook, Twitter, and Netflix for managing large-scale data systems.

2. Apache HBase

Apache HBase is a column-family database built on the Hadoop framework. It is designed for real-time data access and is suitable for storing and analyzing large volumes of structured data. HBase was inspired by Google's Bigtable and is often used in applications that require quick reads and writes on big datasets.

3. Google Cloud Bigtable

Google Cloud Bigtable is a fully managed NoSQL column-family database provided by Google. It is designed for high performance and scalability, making it ideal for handling large volumes of data. It is used in core Google services such as Search, Analytics, and Maps, where fast and efficient data processing is essential.

5.2.5 Graph Based Databases

A graph-based database is a type of NoSQL database that uses a graph structure to store, manage, and query data. Instead of tables and rows (like in relational databases), it represents data as nodes (entities) and edges (relationships). This makes it well-suited for applications that require managing and analyzing complex relationships.

Imagine you are building a social media platform where users can follow each other, like posts, and join groups. A graph database helps store this data in a way that naturally represents these connections.

In this system:

- ◆ Each user (like James, Helen, and Charlie) is stored as a node.
- ◆ Actions such as following another user or liking a post are stored as edges (connections between nodes).

For example, if James follows Helen, there will be a direct connection between them labelled "follows." Similarly, if James likes a post, there will be a connection from James to the post labeled "likes" as shown in figure 5.2.2.

A graph database like Neo4j makes it easy to find relationships without needing complicated searches. If you want to know who Helen follows, the database can quickly retrieve that information because the connections are directly stored. This makes graph databases useful for handling social networks, recommendations, and other relationship-based data.

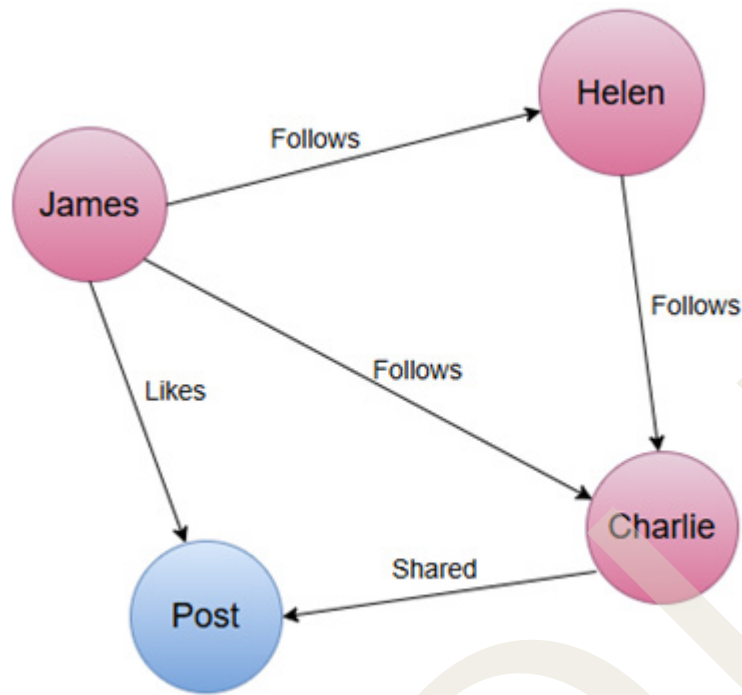


Fig. 5.2.2 Graph Based database

5.2.5.1 Features of Graph based database

1. Node and Edge Structure

Graph databases represent data using nodes and edges. Nodes store entities such as people, products, or places, while edges represent relationships between those entities. This structure replaces the traditional use of tables and rows in relational databases.

2. Schema-Free

Graph databases do not require a fixed schema. This flexibility allows users to easily introduce new types of relationships or properties without redesigning the entire database. It supports dynamic and evolving data models.

3. Highly Connected Data

Graph databases are designed to manage highly connected data. They can efficiently store and query complex relationships, making them ideal for use cases where connections between data points are important.

4. Scalability

These databases can manage large volumes of interconnected data across distributed systems. Their design allows them to scale horizontally, making them suitable for big data applications.

5. Real-World Applications

Graph databases are widely used in real-world scenarios such as fraud detection, recommendation engines, social networking platforms, and knowledge graphs. Their

ability to explore relationships quickly and accurately makes them valuable in these domains.

5.2.5.2 Examples of Graph-based Database

1. Neo4j

Neo4j is one of the most widely used graph databases. It is designed to store and manage highly connected data using nodes and relationships. It supports the Cypher query language, which is specifically created for working with graph structures. Neo4j is used in applications like recommendation systems and fraud detection.

2. Amazon Neptune

Amazon Neptune is a fully managed graph database service provided by Amazon Web Services (AWS). It supports both property graph models (using Gremlin) and RDF graph models (using SPARQL). Neptune is designed for high-performance applications that require fast and reliable querying of complex relationships.

3. ArangoDB

ArangoDB is a multi-model database that supports graph, document, and key-value data models in one system. Its graph capabilities make it suitable for use cases where data relationships are important, while its support for other models adds flexibility in handling various types of data.

Recap

NoSQL

- ◆ Designed for handling large unstructured/semi-structured data.
- ◆ More flexible and scalable than traditional relational databases.
- ◆ Four main types:
 - Key-Value Stores
 - Document-Oriented Databases
 - Column-Family Stores
 - Graph-Based Databases

Key-Value Stores

- ◆ Store data as key-value pairs (like a dictionary).
- ◆ Fast data retrieval using unique keys.
- ◆ Supports various data types (text, numbers, JSON, XML).
- ◆ Examples: Redis, Amazon DynamoDB, Memcached

Document-Oriented Databases

- ◆ Store data as documents (JSON, BSON, XML).
- ◆ Schema-less structure allows flexible data storage.
- ◆ No need for foreign keys; each document is independent.
- ◆ Examples: MongoDB, CouchDB, Firebase Firestore, Amazon DocumentDB

Column-Family Stores

- ◆ Organize data into column families instead of traditional tables.
- ◆ Efficient for big data applications and analytics.
- ◆ Improves query speed by storing related data together.
- ◆ Examples: Apache Cassandra, Apache HBase, Google Cloud Bigtable

Graph-Based Databases

- ◆ Store data as nodes (entities) and edges (relationships).
- ◆ Ideal for applications with complex relationships (social networks, fraud detection).
- ◆ No fixed schema, allowing flexible data connections.
- ◆ Examples: Neo4j, Amazon Neptune, ArangoDB

Objective Type Questions

1. What type of database does not use tables and schemas?
2. Which NoSQL database type stores data as key-value pairs?
3. Which NoSQL database type stores data in document format?
4. Which NoSQL database type organizes data into column families?
5. Which NoSQL database type represents data as nodes and edges?
6. Name a column-family store database inspired by Google Bigtable.
7. Name a graph database service provided by AWS.
8. Name a multi-model database that supports key-value, document, and graph data.

9. Which NoSQL database is designed for mobile and web applications?
10. Which NoSQL database is commonly used for caching frequently accessed data?

Answers to Objective Type Questions

1. NoSQL
2. Key-Value
3. Document-Oriented
4. Column-Family
5. Graph
6. HBase
7. Neptune
8. ArangoDB
9. Firestore
10. Memcached

Assignments

1. Explain the main differences between NoSQL databases and traditional relational databases.
2. Describe the structure of a key-value store database with an example.
3. What are document-oriented databases? Explain their key features.
4. How does a column-family store organize data? Give a simple example.
5. What are graph-based databases? Explain their components with an example.
6. You are tasked with designing a recommendation engine for an e-commerce platform that suggests products to users based on their browsing history, purchase patterns, and social connections (e.g., similar interests or behaviors of other users). Which type of NoSQL database would be the most suitable for implementing this recommendation engine? Justify your choice by explaining.

Suggested Reading

1. Andreas, M., & Kaufmann, M. (2019). *SQL & NoSQL databases: Models, languages, consistency options and architectures for big data management*.
2. Bradshaw, S., Brazil, E., & Chodorow, K. (2019). *MongoDB: The definitive guide: Powerful and scalable data storage*. O'Reilly Media.
3. Hillar, G. C., & Yöndem, D. (2018). *Guide to NoSQL with Azure Cosmos DB: Work with the massively scalable Azure database service with JSON, C#, LINQ, and .NET Core 2*. Packt Publishing.
4. Sullivan, D. (2015). *NoSQL for mere mortals*. Addison-Wesley Professional.

Reference

1. <https://archive.nptel.ac.in/courses/106/104/106104135/>
2. <https://www.geeksforgeeks.org/dbms/types-of-nosql-databases/>

Unit 3

NoSQL Database Examples

Learning Outcomes

Upon the completion of this unit, the learners will be able to:

- ◆ understand the basic concepts of MongoDB, Cassandra, HBase and Neo4j
- ◆ discuss the key features, advantages of different NoSQL Databases
- ◆ describe the working of MongoDB, Cassandra, HBase and Neo4j
- ◆ list the data types of each NoSQL Databases
- ◆ compare the features of NoSQL Databases

Prerequisites

It is essential to comprehend the distinct designs and ideal use cases of NoSQL databases such as MongoDB, Cassandra, HBase, and Neo4j in order to implement them. Designed for applications that need dynamic schemas and hierarchical data structures, MongoDB is a document-oriented database that stores data as adaptable, JSON-like documents. Its schema-less architecture enables quick development and iteration, especially in situations where data models change regularly. A wide-column store built for scalability and high availability, Cassandra, on the other hand, is excellent at managing massive amounts of structured data across dispersed systems. Applications like real-time analytics and e-commerce platforms that cannot afford downtime can benefit from its fault-tolerant architecture.

Built on top of the Hadoop Distributed File System (HDFS), HBase is best suited for applications that need to access big datasets in real time and involve a lot of reading and writing. It works well in situations like online log analytics and huge data processing, and it can handle sparse data. Neo4j is a graph database designed for handling linked data, which makes it perfect for applications where knowing the links between entities is essential, such as fraud detection, recommendation systems, and social networks. Each of these databases has unique benefits, and the best choice will rely on the particular needs of the application, data formats, and scalability.

Keywords

MongoDB, Cassandra, HBASE, Neo4j



Discussion

5.3.1 Overview of MongoDB

MongoDB is an open-source document-oriented database that is designed to store large amounts of data and allows us to work with that data efficiently. It is categorized under the NoSQL database because the storage and retrieval of data in the MongoDB are not in the form of tables. The MongoDB database is developed and managed by MongoDB, Inc. under SSPL(Server Side Public License) and initially released in February 2009. Commonly used for big data applications, content management, and real-time analytics.

MongoDB is a well-known NoSQL database that uses BSON (Binary JSON), a versatile format similar to JSON, to store data. MongoDB makes it simpler to work with unstructured or semi-structured data by using documents and collections rather than tables and rows, as conventional databases do. It is perfect for applications that change quickly because of its flexible schema, which stores several data types in the same collection without worrying about established patterns.

High speed, scalability, and user-friendliness are features of MongoDB. In order to provide quick data access, high availability, and horizontal scaling, it includes capabilities like indexing, replication, and sharding. Web applications, content management systems, real-time analytics, and more all make extensive use of MongoDB because of its integrated capabilities for managing massive volumes of data and enabling sophisticated searches.

It also provides official driver support for all the popular languages like C, C++, C#, and .Net, Go, Java, Node.js, Perl, PHP, Python, Motor, Ruby, Scala, Swift, Mongoid. So, we can create an application using any of these languages. Nowadays, there are so many companies that use MongoDB, like Facebook, Nokia, eBay, Adobe, Google, etc., to store their large amounts of data.

5.3.1.1 Key Features

1. **Document-Oriented Storage:** MongoDB stores data in JSON-like documents called BSON, which allows for flexible and dynamic schemas. This approach makes it easy to model complex and hierarchical data structures without predefined tables.
2. **Scalability:** MongoDB supports horizontal scaling through sharding, where data is distributed across multiple servers. This makes it ideal for handling large datasets and high-throughput applications by improving performance and storage capacity.
3. **High Performance:** Its non-relational architecture helps minimize latency and enhance overall efficiency.
4. **Indexing:** MongoDB provides various indexing options, including single-field, compound, text, and geospatial indexes. Indexing improves query performance by allowing quick searches and sorting of data.

5. **Aggregation Framework:** It allows operations like filtering, grouping, sorting, reshaping, and analyzing data efficiently.
6. **Replication:** MongoDB ensures high availability and redundancy using replica sets, where multiple copies of data are maintained across servers. Automatic failover and data recovery enhance reliability and prevent data loss.
7. **ACID Transactions:** MongoDB supports multi-document ACID transactions, ensuring data consistency across complex operations. This feature provides reliable support for critical financial and e-commerce applications.
8. **Ad-hoc Queries:** Developers can easily retrieve and manipulate data without extensive restructuring.
9. **File Storage (GridFS):** It breaks files into smaller parts, making it suitable for efficient data management and streaming.
10. **Security:** MongoDB offers robust security features like authentication, authorization, role-based access control, and encryption. These measures ensure data protection, preventing unauthorized access and safeguarding sensitive information.

5.3.1.2 Data Model

MongoDB uses a document-oriented, schema-less model that stores data as JSON-like documents (BSON format). This approach provides flexibility for handling structured, semi-structured, and unstructured data, making it highly adaptable to changing application requirements. The key components are:

1. **Database:** A physical container for collections.
2. **Collection:** A group of related documents (similar to a table in relational databases).
3. **Document:** A record in BSON format (similar to a row). This is made of fields. It is similar to a tuple in RDBMS, but it has a dynamic schema. Documents of the same collection need not have the same set of fields. The MongoDB document structure is illustrated below (Figure: 5.3.1).

A simple MongoDB document Structure:

```
{  
  title: 'COMPUTER SCIENCE',  
  by: 'Harshit Gupta',  
  url: 'https://www.studies.org',  
  type: 'NoSQL'  
}
```

Fig. 5.3.1 A simple MongoDB document Structure

4. **Field:** A key-value pair within a document (similar to a column). The following Table:5.3.1 shows the relationship of RDBMS terminology with MongoDB.

Table 5.3.1 Relationship of RDBMS and MongoDB

RDBMS	MongoDB	Description
Database	Database	Stores multiple collections.
Table	Collection	Group related documents.
Tuple/Row	Document	A BSON object containing key-value pairs.
Column	Field	Stores data attributes within documents.

5.3.1.3 Working of MongoDB

MongoDB is referred to as a NoSQL database because it offers a completely alternative mechanism for storing and retrieving data, rather than being based on the table-like relational database structure. Before moving on to how it operates, let's first go over some of MongoDB's key components.

Drivers: Drivers are present on your server that are used to communicate with MongoDB. The drivers support by the MongoDB are C, C++, C#, and .Net, Go, Java, Node.js, Perl, PHP, Python, Motor, Ruby, Scala, Swift, Mongoid.

MongoDB Shell: Also known as mongo shell, this JavaScript interface is interactive and supports MongoDB. It is utilized for administrative tasks as well as inquiries and data modifications.

Storage Engine: The Storage Engine is a crucial component of MongoDB, which is often used to control the way data is kept on disk and in memory. Several search engines may be used with MongoDB.

The way MongoDB functions is depicted in Figure 5.3.2. MongoDB work in two layers:

1. Application Layer
2. Data Layer

The **Application Layer**, often referred to as the Final Abstraction Layer, is composed of two components: the Frontend (User Interface) and the Backend (server). The frontend is where the user interacts with MongoDB via a mobile device or web browser. online sites, mobile applications, Android default apps, iOS apps, and more are included in this mobile and online platform. In addition to a server for server-side functionality, the backend includes drivers or a mongo shell for query-based communication with the MongoDB server.

The **Data Layer's MongoDB** server receives these requests. The queries are now sent to the storage engine by the MongoDB server. The MongoDB server itself does not read or write data directly to disk, memory, or files. After passing the received queries to the storage engine, the storage engine is responsible to read or write the data in the files or memory, basically it manages the data.

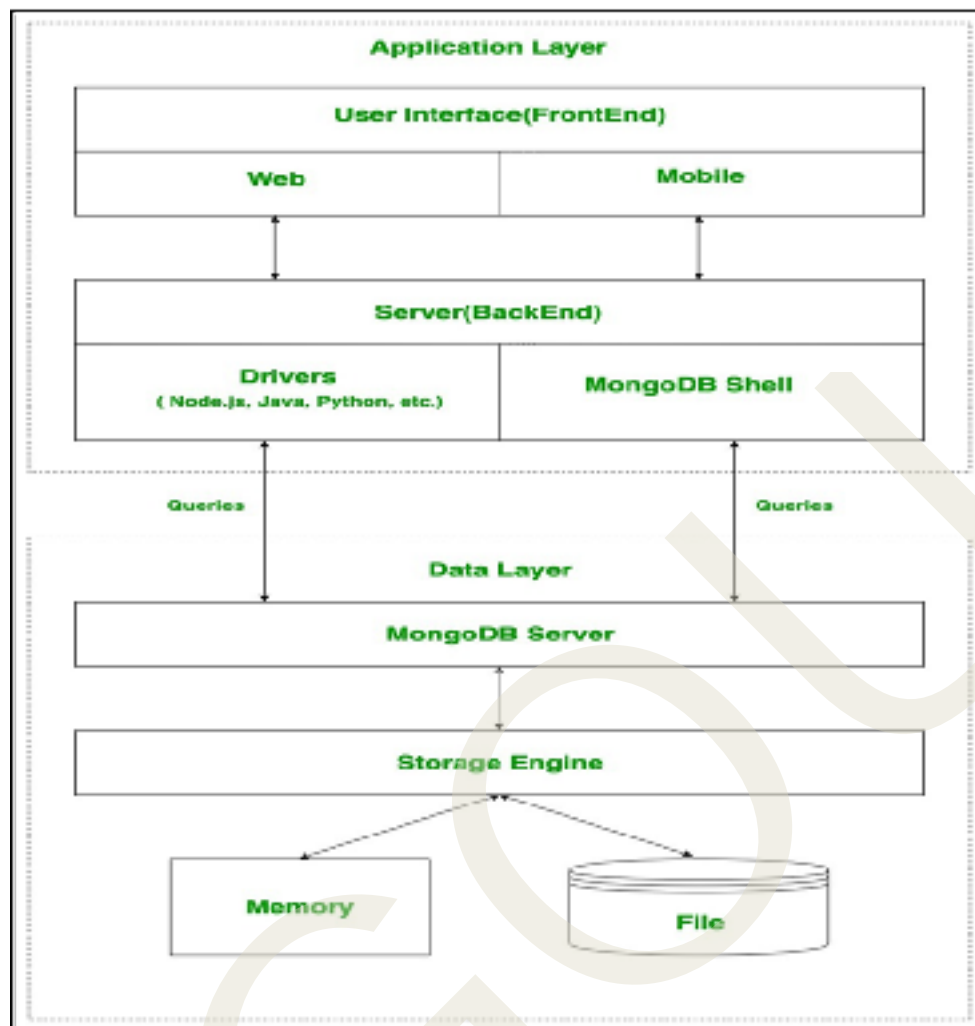


Fig. 5.3.2 Working of MongoDB

5.3.1.4 Advantages of MongoDB

1. **Schema Flexibility:** MongoDB uses a schema-less design, allowing documents (JSON-like objects) to have different structures within a collection.
2. **Scalability and High Performance:** Supports horizontal scaling through sharding, allowing databases to grow across multiple servers.
3. **High Availability with Replica Sets:** Provides automatic failover and redundancy with replica sets. Also ensures data availability even if some nodes fail.
4. **Rich Query Language:** Supports powerful, expressive queries for filtering, sorting, aggregating, and transforming data.
5. **Ease of Use and Development:** Integrates well with modern tech stacks, especially with JavaScript, Node.js, and Python.
6. **Security Features:** Supports Role-Based Access Control (RBAC), SSL/TLS encryption, and authentication mechanisms.

7. **Automatic Scaling and Load Balancing:** Handles massive datasets with ease by distributing data across multiple shards
8. **Aggregation Framework:** Pipelines allow for chaining multiple operations to transform and aggregate data effectively.

5.3.1.5 Data Types

MongoDB stores its documents in BSON, which is a binary encoded version of JSON. We can use BSON to run remote procedures in MongoDB. The BSON format is compatible with a number of data types. Arrays and strings are among the many data types that MongoDB offers. The most popular MongoDB data types are shown here, along with useful examples to help clarify them.

1. String

This is the most commonly used data type in MongoDB to store data, BSON strings are of UTF-8. So, the drivers for each programming language convert from data types to the string format of the language to UTF-8 while serializing and de-serializing BSON. The string must be a valid UTF-8.

e.g.: { "name": "Akash" }

2. Integer

In MongoDB, the integer data type is used to store an integer value. We can store integer data types in two forms: 32-bit signed integer and 64-bit signed integer. These are used to store whole numbers, such as ages, counts, or any other numerical data that doesn't require decimal points.

e.g.: { "age": 20 }

3. Double

The double data type is used for storing floating-point numbers (decimal values). It's commonly used for storing data that requires decimal precision, such as prices, percentages, or scores.

e.g.: { "marks": 546.43 }

4. Boolean

The boolean data type stores one of two values: true or false. It's used for representing binary states, such as "active/inactive" or "pass/fail."

e.g.: { "pass": true }

5. Null

The null data type stores a null value. This is useful when you want to represent the absence of data, such as an optional field that may not be set.

e.g.: { "mobile": null }

6. Array

The array data type allows us to store multiple values in a single field. MongoDB arrays can contain values of the same or different data types, providing flexibility in how you store collections of related data. In MongoDB, the array is created using square brackets([]).

e.g.: { "skills": ["c", "c++", "java", "python"] }

7. Object (Embedded Document)

Object data type stores embedded documents. Embedded documents are also known as nested documents. Embedded documents or nested documents are those types of documents which contain a document inside another document. Embedded documents allow us to structure our data hierarchically, which is useful for representing more complex data models.

e.g.: { "book":
 { "name": "C++",
 "Writer": "Akash" }
}

8. Object Id

Whenever we create a new document in the collection MongoDB automatically creates a unique object id for that document (if the document does not have it). There is an `_id` field in MongoDB for each document. The data which is stored in Id is of hexadecimal format and the length of the id is 12 bytes which consist:

- ◆ 4-bytes for Timestamp value.
- ◆ 5-bytes for Random values. i.e., 3-bytes for machine Id and 2-bytes for process Id.
- ◆ 3- bytes for Counter.

e.g.: { "_id": ObjectId("601af71f6fd54aa34c9c6df9") }

9. Undefined

The undefined data type represents a value that is not defined. It is rarely used in modern MongoDB applications, as it has been replaced by the null value for most practical purposes.

e.g.: { "duration": undefined }

10. Binary Data

The binary data data type stores binary information such as images, files, or encrypted data. Binary data is often used when working with non-textual data.

e.g.: { "binaryValue": "110011001" }



11. Date

Date data type stores date. It is a 64-bit integer which represents the number of milliseconds. BSON data type generally supports UTC datetime and it is signed. If the value of the date data type is negative then it represents the dates before 1970. There are various methods to return a date, it can be returned either as a string or as a date object. Some methods (Figure:5.3.3) for the date are:

- ◆ **Date():** It returns the current date in string format.
- ◆ **new Date():** Returns a date object. Uses the ISODate() wrapper.
- ◆ **new ISODate():** It also returns a date object. Uses the ISODate() wrapper.

```
> var date1 = Date()
> var date2 = new Date()
> var date3 = new ISODate()
> db.date.insertOne({Date1:date1,Date2:date2,Date3:date3})
{
  'acknowledged' : true,
  'insertedId' : ObjectId("601afa326fd54aa34c9c6dfc")
}
> db.date.find().pretty()
{
  "_id" : ObjectId("601afa326fd54aa34c9c6dfc"),
  "Date1" : "Thu Feb 04 2021 01:00:34 GMT+0530 (India Standard Time)",
  "Date2" : ISODate("2021-02-03T19:30:40.014Z"),
  "Date3" : ISODate("2021-02-03T19:30:55.454Z")
}
```

Fig. 5.3.3 Various methods for Date

5.3.1.6 Operations

MongoDB, a popular NoSQL database, facilitates efficient storage and retrieval of data through its CRUD operations: Create, Read, Update, and Delete. Below are simple examples of each operation, along with explanations and expected outputs.

1. Create Operations

The create or insert operations are used to insert or add new documents in the collection. If a collection does not exist, then it will create a new collection in the database. We can perform, create operations using the following methods provided by MongoDB (Table:5.3.2).

Table 5.3.2 Create Operation methods provided by MongoDB

Method	Description
db.collection.insertOne()	It is used to insert a single document in the collection.
db.collection.insertMany()	It is used to insert multiple documents in the collection.

A. Inserting a Single Document

db.collection.insertOne() inserts a single document into the specified collection. The document is a JSON-like object containing name, age, and city fields in the following example (Table:5.3.3).

Syntax:

```
db.collection.insertOne({  
    field1: value1,  
    field2: value2, // Additional fields as needed  
});
```

Table 5.3.3 Example and Output of Inserting a Single Document

Example	Output
<pre>db.collection.insertOne({ name: "Alice", age: 30, city: "New York" });</pre>	<pre>{ "acknowledged": true, "insertedId": ObjectId("60c72b2f5b9c1b3d4f8e4e3f") }</pre>

B. Inserting Multiple Documents

db.collection.insertMany() inserts multiple documents into the collection. An array of JSON-like objects is provided, each representing a document is shown in below example (Table:5.3.4).

Syntax:

```
db.collection.insertMany([  
    { field1: value1, field2: value2 },  
    { field1: value3, field2: value4 }, // Additional documents as needed  
]);
```

Table 5.3.4 Example and Output of Inserting Multiple Documents

Example	Output
<pre>db.collection.insertMany([{ name: "Bob", age: 25, city: "Los Angeles" }, { name: "Charlie", age: 35, city: "Chicago" }]);</pre>	<pre>{ "acknowledged": true, "insertedIds": [ObjectId("60c72b2f5b9c1b3d4f8e4e40"), ObjectId("60c72b2f5b9c1b3d4f8e4e41")]}</pre>

2. Read Operations

The Read operations are used to retrieve documents from the collection, or in other words, read operations are used to query a collection for a document. We can perform a read operation using the following methods provided by MongoDB (Table:5.3.5).

Table 5.3.5 Read operation methods provided by MongoDB

Method	Description
db.collection.find()	It is used to retrieve documents from the collection.
db.collection.find({ field1: value })	Retrieving documents with a specific condition
db.collection.findOne()	Retrieves a single document that matches the query criteria.

A. Retrieving All Documents

db.collection.find() used to retrieve documents from the collection. The following example (Table:5.3.6) shows the output of this function while it is used.

Syntax: db.collection.find();

Table 5.3.6 Example and Output of Retrieving All Documents

Example	Output
db.collection.find();	<pre>{ "_id": ObjectId("60c72b2f5b9c1b3d4f8e4e3f"), "name": "Alice", "age": 30, "city": "New York" } { "_id": ObjectId("60c72b2f5b9c1b3d4f8e4e40"), "name": "Bob", "age": 25, "city": "Los Angeles" } { "_id": ObjectId("60c72b2f5b9c1b3d4f8e4e41"), "name": "Charlie", "age": 35, "city": "Chicago" }</pre>

B. Retrieving Documents with Specific Criteria

db.collection.find({ field1: value }) retrieves documents with a specific condition. In the following example (Table:5.3.7) ***db.collection.find({ age: { \$gt: 30 } })*** retrieves documents where the age field is greater than 30.

Syntax: *db.collection.find({ field1: value });*

Table 5.3.7 Example and Output of Retrieving Documents with Specific Criteria

Example	Output
<i>db.collection.find({ age: { \$gt: 30 } });</i>	<i>{ "_id": ObjectId("60c72b2f5b9c1b3d4f8e4e41"), "name": "Charlie", "age": 35, "city": "Chicago" }</i>

C. Retrieving a Single Document

db.collection.findOne() retrieves a single document that matches the query criteria. Consider a students collection with the following documents. To retrieve the first student named "Bob" by the below code (Table:5.3.8).

Syntax: *db.collection.findOne({ field1: value });*

Table 5.3.8 Example and Output of Retrieving a Single Document

Example	Output
<i>db.students.findOne({ name: "Bob" });</i>	<i>{ "_id": 2, "name": "Bob", "age": 22, "major": "Mathematics" }</i>

3. Update Operations

The update operations are used to update or modify the existing document in the collection. We can update a single document or multiple documents that match a given query. We can perform update operations using the following methods (Table:5.3.9) provided by MongoDB.

Table 5.3.9 Update operations provided by MongoDB

Method	Description
<i>db.collection.updateOne()</i>	It is used to update a single document in the collection that satisfies the given criteria.
<i>db.collection.updateMany()</i>	It is used to update multiple documents in the collection that satisfy the given criteria.
<i>db.collection.replaceOne()</i>	It is used to replace a single document in the collection that satisfies the given criteria.

A. Updating a Single Document

db.collection.updateOne() used to update a single document in the collection that satisfies the given criteria. Example and output of this criteria is shown in Table 5.3.10.



Syntax:

```

db.collection.updateOne(
    { field1: value }, // Filter criteria
    { $set: { field2: newValue } } // Update operation
);

```

Table 5.3.10 Example and Output of Updating a Single Document

Example	Output
<pre> db.students.updateOne({ name: "Alice" }, { \$set: { major: "Data Science" } }); </pre>	<pre> { "acknowledged": true, "matchedCount": 1, "modifiedCount": 1 } </pre>

B. Updating Multiple Documents

db.collection.updateMany() is used to update multiple documents in the collection that satisfy the given criteria. The following Table 5.3.11 shows an example and output of updating multiple documents.

Syntax:

```

db.collection.updateMany(
    { field1: value }, // Filter criteria
    { $set: { field2: newValue } } // Update operation
);

```

Table 5.3.11 Example and Output of Updating Multiple Documents

Example	Output
<pre> db.students.updateMany({ age: { \$gt: 23 } }, { \$set: { status: "Alumni" } }); </pre>	<pre> { "acknowledged": true, "matchedCount": 2, "modifiedCount": 2 } </pre>

C. Replacing a Document

db.collection.replaceOne() used to replace a single document in the collection that satisfies the given criteria. Table 5.3.12 shows the example and output of replacing a document.

Syntax:

```
db.collection.replaceOne(  
    { field1: value }, // Filter criteria  
    { field1: value, field2: newValue, // Additional fields  
    } );
```

Table 5.3.12 Example and Output of Replacing a Document

Example	Output
<code>db.students.replaceOne({ name: "Alice" }, { name: "Alice", age: 22, major: "Data Science" });</code>	<code>{ "acknowledged": true, "matchedCount": 1, "modifiedCount": 1 }</code>

4. Delete Operations

The delete operation is used to delete or remove the documents from a collection. We can delete documents based on specific criteria or remove all documents. We can perform delete operations using the following methods provided by MongoDB (Table:5.3.13).

Table 5.3.13 Delete operation methods provided by MongoDB

Method	Description
<code>db.collection.deleteOne()</code>	It is used to delete a single document from the collection that satisfies the given criteria.
<code>db.collection.deleteMany()</code>	It is used to delete multiple documents from the collection that satisfy the given criteria.

A. Deleting a Single Document

db.collection.deleteOne() is used to delete a single document from the collection that satisfies the given criteria. Table 5.3.14 shows the example and output of deleting a single document.

Syntax: `db.collection.deleteOne({ field1: value });`

Table 5.3.14 Example and Output of Deleting a Single Document

Example	Output
<code>db.students.deleteOne({name: "Charlie" });</code>	<pre>{ "acknowledged": true, "deletedCount": 1 }</pre>

B. Deleting Multiple Documents

db.collection.deleteMany() is used to delete multiple documents from the collection that satisfy the given criteria. Table 5.3.15 shows the example and output of deleting a multiple document.

Syntax: `db.collection.deleteMany({ field1: value });`

Table 5.3.15 Example and Output of Deleting Multiple Documents

Example	Output
<code>db.students.deleteMany({ age: { \$lt: 23 } });</code>	<pre>{ "acknowledged": true, "deletedCount": 1 }</pre>

5.3.2 Overview of Cassandra

Apache Cassandra is a powerful open-source NoSQL database designed to handle large amounts of data across many servers with high availability and no single point of failure. It uses a peer-to-peer architecture where all nodes are equal, allowing for easy scaling by simply adding more nodes. Cassandra is known for its high write throughput, making it ideal for write-heavy workloads. It stores data in tables with rows identified by primary keys, and uses a write-optimized system involving commit logs, and SSTables (Sorted String Table) for durability and efficiency.

Cassandra is commonly used in applications like IoT platforms, messaging systems, and real-time analytics that require massive scalability and continuous availability. Its tunable consistency model allows developers to balance consistency, availability, and performance based on their needs. While it excels at handling large-scale, distributed data, it has limitations when it comes to complex queries and fully supporting ACID transactions. However, its speed, reliability, and ability to scale horizontally make it a popular choice for modern, high-performance applications.

5.3.2.1 Key Features

Cassandra's unique combination of decentralization, data replication, tunable consistency, and support for flexible data models makes it ideal for real-time big data applications, especially those that span geographically distributed locations. The following key features illustrate why Cassandra is well-suited for modern, data-intensive environments.

1. **Decentralized Architecture:** All nodes are equal, with no single point of failure. Data is distributed evenly across the cluster, ensuring high availability and fault tolerance.
2. **Scalability:** Provides horizontal scalability by allowing easy addition of new nodes without downtime, supporting massive datasets across many servers.
3. **High Availability:** Automatic data replication across multiple nodes and data centers ensures data accessibility even if some nodes fail.
4. **Tunable Consistency:** Offers adjustable consistency levels (e.g., ONE, QUORUM, ALL) to balance between consistency, availability, and performance based on application needs.
5. **High Write Performance:** Uses a write-optimized architecture involving commit logs, memtables, and SSTables, providing fast write operations ideal for write-heavy workloads.
6. **Flexible Data Model:** Supports schema-free, semi-structured, and structured data using tables with rows defined by primary keys (partition and clustering keys).
7. **Cassandra Query Language (CQL):** Provides a SQL-like language for querying and managing data, making it easier for developers to work with.
8. **Eventual Consistency:** Uses techniques like hinted handoff, read repair, and anti-entropy repair to achieve consistency over time without impacting availability.
9. **Multi-Data Center Replication:** Ensures high availability and low latency by replicating data across geographically distributed data centers.
10. **Fault Tolerance and Durability:** Ensures data durability through mechanisms like commit logs and data replication, providing resilience against hardware failures.

5.3.2.2 Data Model

The Cassandra data model is designed for high availability, scalability, and efficient write operations. It uses a column-oriented, schema-free, flexible model that suits semi-structured or structured data. The key components are:

1. **Keyspace** (Equivalent to a database in RDBMS): The top-level namespace for tables. Defines properties like replication strategy and replication factor.



Example:

```
CREATE KEYSPACE my_keyspace
```

```
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```

- 2. Tables (Column Families):** The primary storage structure containing rows and columns. Schema definition is required, but it's flexible and allows adding columns as needed.

Example:

```
CREATE TABLE users (  
    user_id UUID PRIMARY KEY,  
    name TEXT,  
    email TEXT,  
    age INT );
```

- 3. Rows:** Each row is identified by a Primary Key. Rows are stored in sorted order based on the Clustering Keys. New rows can be added without altering the schema.
- 4. Columns:** Basic data units identified by names and containing values. Can be added dynamically within rows of the same table.
- 5. Primary Key:** Uniquely identifies each row. Partition Key: The first part of the primary key used to distribute data across nodes. Clustering Key(s): Define the sort order of data within a partition.

Example:

```
CREATE TABLE messages (  
    user_id UUID,  
    message_id TIMEUUID,  
    content TEXT,  
    timestamp TIMESTAMP,  
    PRIMARY KEY (user_id, message_id));
```

Here, **user_id** is the **Partition Key** and **message_id** is the **Clustering Key**.

5.3.2.3 Working of Cassandra

Avinash Lakshman and Prashant Malik initially developed Cassandra at Facebook to power the Facebook inbox search feature. Facebook released Cassandra as an open source project on google code in July 2008. Cassandra powers online services and mobile backend for some of the world's most recognizable brands, including Apple, Netflix, and Facebook. In this section we will describe the following component of Apache Cassandra.

A. Basic Terminology: Understanding Cassandra requires knowing some foundational terms. Here are the most important ones.

- ◆ Node
- ◆ Data Center
- ◆ Cluster

1. **Node:** Node is the basic component in Apache Cassandra. It is the place where actual data is stored. For example, as shown in figure 5.3.4 nodes which has IP address 10.0.0.7 contain data (keyspace which contain one or more tables).

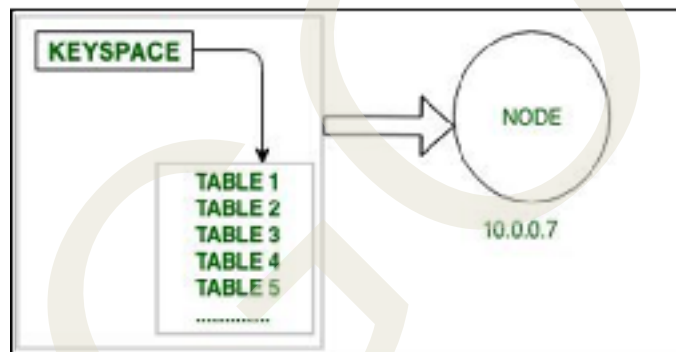


Fig. 5.3.4 Representation of Node

2. **Data Center:** Data Center is a collection of nodes.

Example:

$$DC = N1 + N2 + N3 \dots$$

DC: Data Center

N1: Node 1

N2: Node 2

N3: Node 3

3. **Cluster:** It is the collection of many data centers. Figure 5.3.5 depicted nodes, data centers and clusters

Example:

$C = DC1 + DC2 + DC3....$

C: Cluster

DC1: Data Center 1

DC2: Data Center 2

DC3: Data Center 3

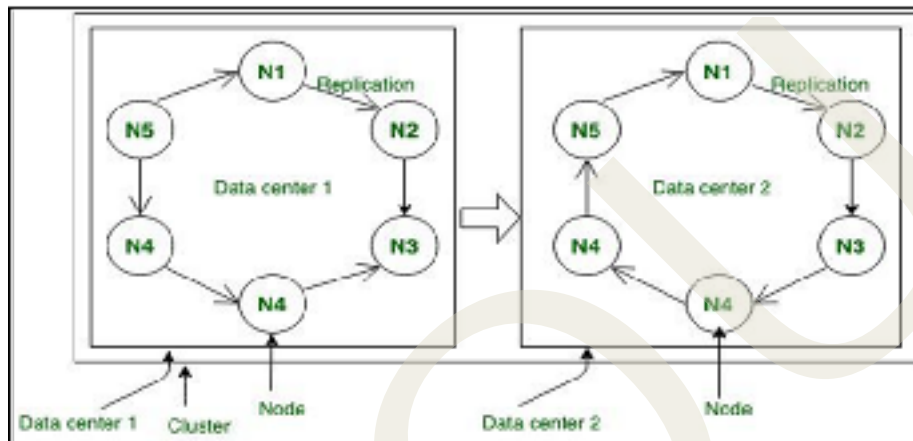


Fig. 5.3.5 Node, Data center and Cluster

B. Operations: It also includes maintenance tasks like replication, compaction, and consistency management. Cassandra mainly consists of two operations;

- ◆ Read Operation
- ◆ Write Operation

1. Read Operation: In Read Operation there are three types of read requests that a coordinator can send to a replica. The node that accepts the write requests is called coordinator for that particular operation.

Step-1: Direct Request: In this operation coordinator node sends the read request to one of the replicas.

Step-2: Digest Request: In this operation coordinator will contact replicas specified by the consistency level.

Step-3: Read Repair Request: If there is any case in which data is not consistent across the node then background Read Repair Request initiated that makes sure that the most recent data is available across the nodes.

2. Write Operation: In Write Operation there are three types of write responses.

Step-1: In Write Operation as soon as we receive a request then it is first dumped into the commit log to make sure that data is saved.

Step-2: Insertion of data into a table that is also written in MemTable that holds the data till it gets full.

Step-3: If MemTable reaches its threshold then data is flushed to SS Table. Figure 5.3.6. depicted the write operation in Cassandra.

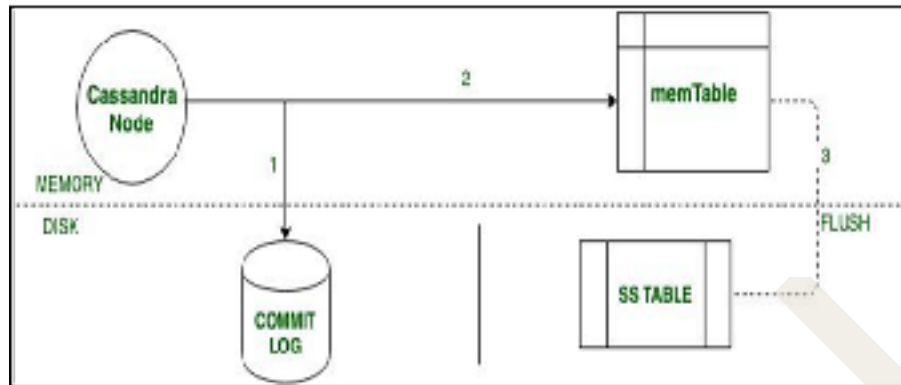


Fig. 5.3.6 Write Operation in Cassandra

C. Storage Engine: A Storage Engine is the underlying software component of a database responsible for managing how data is stored, retrieved, and maintained on disk or memory. It handles essential tasks like writing data, reading data, indexing, compression, caching, and ensuring durability. It consists of:

- ◆ CommitLog
- ◆ Memtables
- ◆ SSTables
- ◆ Data Replication Strategies

1. **Commit log:** Commit log is the first entry point while writing to disk or memTable. The purpose of commit log in apache Cassandra is to server sync issues if a data node is down.
2. **Mem-table:** After data written in Commit log then after that data is written in Mem-table. Data is written in Mem-table temporarily.
3. **SSTable:** Once Mem-table will reach a certain threshold then data will flushed to the SSTable disk file.
4. **Data Replication Strategy:** Basically it is used for backup to ensure no single point of failure. In this strategy Cassandra uses replication to achieve high availability and durability.

5.3.2.4 Advantages of Cassandra

1. **High Availability and Fault Tolerance:** Decentralized architecture with no single point of failure. Data is automatically replicated across multiple nodes for redundancy.
2. **Scalability:** Easily add more nodes to the cluster without downtime. Designed to handle large amounts of data and high write/read throughput.

3. **Fast Write Performance:** Write operations are highly efficient due to Commit Log and Memtable mechanisms. Ideal for applications requiring high-speed data ingestion.
4. **Flexible Data Model:** Schema-optional model allows dynamic addition of columns. Supports structured, semi-structured, and unstructured data.
5. **Distributed and Decentralized:** Every node in the cluster is equal (peer-to-peer). Automatic load balancing using Partition Keys.
6. **Linear Performance Scaling:** Performance increases linearly with the addition of nodes. No bottlenecks caused by master nodes.
7. **Efficient Data Distribution:** Partitioning and Clustering Keys ensure effective data distribution and sorting.
8. **Built for High Availability Applications:** Commonly used for IoT, messaging systems, social networks, and real-time analytics.

5.3.2.5 Data Types

Cassandra supports a wide range of built-in data types that allow developers to model complex and scalable applications effectively. These data types include both primitive types like int, text and boolean, as well as collection types such as list, set, and map, which enable the storage of multiple values within a single column. Additionally, Cassandra offers user-defined types (UDTs) to support the creation of custom, reusable data structures. Understanding these data types is essential for designing efficient schemas, ensuring data consistency, and optimizing performance in large-scale deployments. Cassandra data types are broadly categorized into:

- A. Primitive Data Types
- B. Collection Data Types
- C. User-Defined Types (UDTs)

A. Primitive Data Types

Primitive data types in Apache Cassandra form the foundational building blocks for defining columns in a table. These basic data types represent single, indivisible values and are used to store common types of data such as numbers, text, and dates. Table: 5.3.16 shows various primitive data types used in Cassandra.

Table 5.3.16 Primitive data types in Cassandra

Data Type	Definition	Syntax	Example
ascii	Strings with ASCII characters.	ascii	'Hello'
bigint	64-bit signed integer.	bigint	1234567890
boolean	Boolean values.	boolean	true / false
date	Stores date (no time).	date	2025-03-14
decimal	Variable-precision decimal.	decimal	10.5
double	64-bit floating-point number.	double	3.14159
int	32-bit signed integer	int	100

B. Collection Data Types

Collection data types in Apache Cassandra allow the storage of multiple related values within a single column, making data modeling more flexible and efficient. These types are especially useful when dealing with one-to-many relationships or grouping multiple values that logically belong together. Cassandra supports three main collection data types (Table:5.3.17): list, which stores ordered elements; set, which holds unique and unordered elements; and map, which stores key-value pairs for quick lookups. By using collections, developers can reduce the need for multiple tables or complex joins, enabling more efficient data access patterns. However, proper usage and size control are important to maintain performance and scalability in large-scale applications.

Table 5.3.17 Collection Data Types in Cassandra

Data Type	Definition	Syntax	Example
list	Ordered collection of elements.	list<text>	['Apple', 'Banana']
set	Unordered collection of unique elements.	set<int>	{1, 2, 3}
map	Key-value pair collection.	map<text: int>	{'Apple': 10, 'Banana': 20}

Examples Using Collection Types

Creating a table:

```
CREATE TABLE students (  
    student_id uuid PRIMARY KEY,  
    name text,  
    subjects list<text>,  
    marks map<text, int> );
```

Inserting data:

```
INSERT INTO students (student_id, name, subjects, marks)  
VALUES (uuid(), 'John', ['Math', 'Science'], {'Math': 85, 'Science': 90});
```

C. User-Defined Types (UDTs)

User-Defined Types (UDTs) in Apache Cassandra provide a powerful way to create custom, structured data types that group multiple related fields into a single column. UDTs enable developers to define complex data models by combining various primitive and collection data types into a reusable structure. For example, an address UDT might include fields like street, city, and zipcode, allowing this group of information to be

stored and retrieved as one logical unit. This feature enhances schema flexibility and readability, especially in scenarios where nesting or encapsulation of related attributes is needed. UDTs help reduce schema complexity and improve data organization without sacrificing Cassandra's performance and scalability advantages.

Example:

```
CREATE TYPE address (  
    street text,  
    city text,  
    zip int);
```

Examples Using UDT

Creating a Table:

```
CREATE TABLE employees (  
    employee_id uuid PRIMARY KEY,  
    name text,  
    contact_address address);
```

Inserting Data:

```
INSERT INTO employees (employee_id, name, contact_address)  
VALUES (uuid(), 'Alice', {street: 'Main St', city: 'New York', zip: 10001});
```

5.3.2.6. Operations

Cassandra operations are mainly centered around table operations like create, insert, truncate, drop, along with maintenance operations to ensure data integrity, consistency, and availability.

A. Creating a table: First, we are going to create a table namely as Register in which we have id, name, email, city are the fields.

Example:

```
Create table Register (  
    id uuid primary key,  
    name text,  
    email text,  
    city text );
```

B. Inserting data: After creating a table now, we are going to insert some data into the Register table.

Example:

```
Insert into Register (id, name, email, city)
values(uuid(), 'Ashish', 'ashish05.rana05@gmail.com', 'delhi');

Insert into Register (id, name, email, city)
values(uuid(), 'abi', 'abc@gmail.com', 'mumbai');

Insert into Register (id, name, email, city)
values(uuid(), 'rana', 'def@gmail.com', 'bangalore');
```

C. Verify the results: To verify the results using the following query.

Example: `select * from Register;`

Output: The result of the above example is shown below.

index	id	city	email	name
0	03ea1a0a-7878-4ecf-a2fd-c4a9bed7361a	delhi	ashish05.rana05@gmail.com	Ashish
1	a4e6d978-7d97-422a-a854-eca89dabdc74	bangalore	def@gmail.com	rana
2	ac4ad238-761d-4cf5-b907-6c4e02b7e3fa	mumbai	abc@gmail.com	abi

D. Updating the table: To update the table use the following query.

Example:

```
update Register set
email = 'abe@gmail.com'
where id = 57280025-1261-44ab-85d4-62ab2c58a1c1;
```

E. Deleting data from table Register: To delete data from table Register used the following query.

Example: `truncate Register;`

F. Delete a table: To delete table schema and data from table Register used the following query.

Example: `drop table Register;`

5.3.3 Overview of HBASE

HBase is an open-source, distributed, non-relational database built on top of the Hadoop Distributed File System (HDFS). It is designed to handle large amounts of structured and semi-structured data, providing real-time read/write access. HBase uses a column-oriented storage model, making it highly efficient for querying sparse data and scaling horizontally by distributing data across multiple servers. Unlike traditional SQL databases, HBase follows a NoSQL approach, offering features like automatic sharding, strong consistency, and high availability.

HBase is particularly useful for applications that require fast, random read and write operations over large datasets, such as time-series data, logs, or messaging systems. It provides a flexible schema where tables are defined with column families rather than fixed columns, enabling efficient storage and retrieval. HBase integrates seamlessly with Hadoop's ecosystem, including MapReduce for batch processing, and supports APIs for Java, REST, and Thrift, making it a popular choice for Big Data applications that demand scalable and high-performance storage solutions.

5.3.3.1 Features

1. **Scalability:** HBase scales horizontally by distributing data across multiple servers (Region Servers) in a cluster. This allows it to handle petabytes of data with ease.
2. **Column-oriented Storage:** Data is stored in a column-family-oriented model, which provides efficient read/write operations for sparse data and allows flexible schema design.
3. **Automatic Sharding:** Tables are automatically divided into regions and distributed across multiple servers, enabling automatic load balancing and high availability.
4. **Strong Consistency:** Unlike many NoSQL databases, HBase provides strong consistency for read and write operations, ensuring data reliability.
5. **High Availability:** Built on top of HDFS, HBase inherits HDFS's fault tolerance and can recover quickly from failures.
6. **Real-time Read/Write Access:** Supports fast, random read/write operations, making it suitable for real-time applications requiring quick data access.
7. **Flexible Data Model:** Supports dynamic schema design where column families are predefined but columns can be added on the fly.
8. **Integrated with Hadoop Ecosystem:** Works seamlessly with Hadoop tools like MapReduce, Hive, and Spark for batch processing, analytics, and querying.

5.3.3.2 Data Model

The HBase data model is designed to handle massive amounts of sparse, semi-structured data. Its structure is based on a column-family-oriented model rather than

the traditional row-column model used in relational databases. HBase Data Model Components are:

1. **Table:** A collection of rows stored in a distributed manner. Each table is defined by its name and a set of column families.
2. **Row:** Each row is uniquely identified by a row key (byte array), which is sorted lexicographically for fast retrieval. Row keys are the only way to access data, making row key design crucial for performance.
3. **Column Family:** A logical grouping of columns. Each table must define at least one column family. Data within a column family is stored together on disk, enhancing read/write efficiency. Column families are defined at schema definition time and rarely change.
4. **Column Qualifier:** The name of a specific column within a column family. Columns are dynamic; they can be created on the fly without modifying the table schema.

Example Data Model: Imagine a Table 5.3.18 named Users with a column family Info:

Table 5.3.18 Users

Row Key	Info:Name	Info:Email	Info:Phone
user001	Alice	alice@example.com	123-456-7890
user002	Bob	bob@example.com	987-654-3210

Table Name: Users

Column Family: Info

Columns: Name, Email, Phone

Row Key: Unique user identifier (e.g., user001, user002).

5.3.3.3 Working

Apache HBase is a distributed, column-oriented NoSQL database built on top of the Hadoop ecosystem, designed to handle large volumes of sparse data. It operates similarly to Google's Bigtable and is optimized for real-time read and write access to big data. HBase stores data in tables with rows and columns, where each row is identified by a unique row key and columns are grouped into column families. Data within HBase is stored in HDFS (Hadoop Distributed File System), providing fault tolerance and scalability. When a user writes data, it is first stored in a write-ahead log and a memory store (MemStore), and later flushed to disk as HFiles. Reads are served by checking the MemStore and HFiles through efficient indexing. HBase supports random, real-time access to data and integrates well with Hadoop's MapReduce for batch processing, making it ideal for use cases that demand both high throughput and low latency. HBase architecture has 3 main components: HMaster, Region Server, Zookeeper. Figure 5.3.7 depicted the Architecture of HBase.

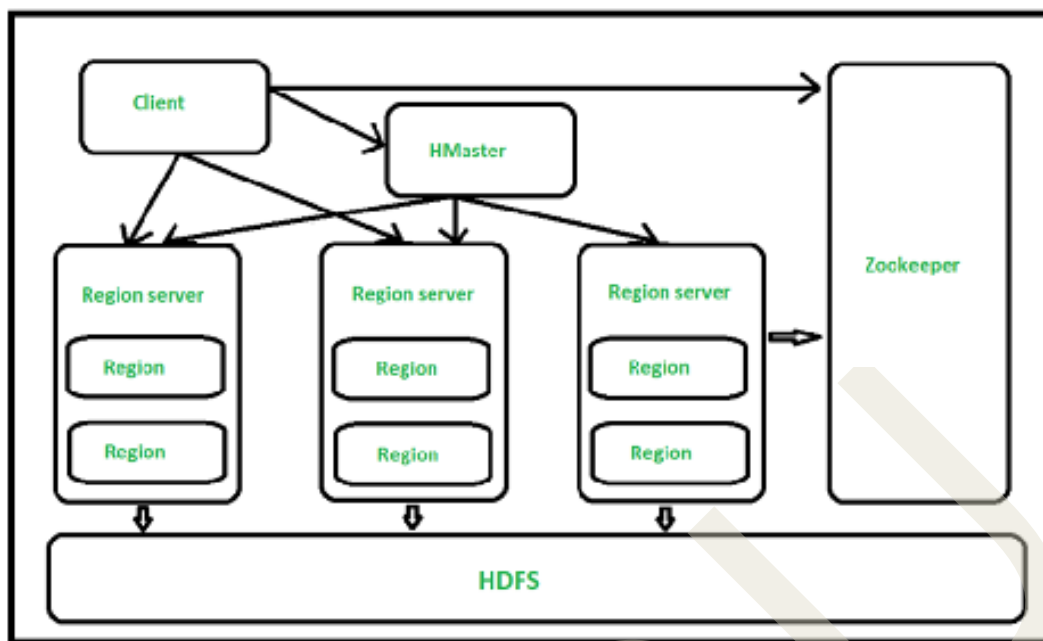


Fig. 5.3.7 Architecture of HBase

1. HMaster

HMaster is the HBase implementation of the Master Server. It's a procedure wherein DDL (create, delete table) operations and regions are allocated to region servers. It keeps an eye on every instance of a Region Server in the cluster. In a distributed setting, Master manages several background threads. Numerous functions of HMaster include failover and load balancing management.

2. Region Server

Regions are the horizontal divisions of HBase tables by row key range. Regions are the fundamental building blocks of an HBase cluster, which are made up of column families and the distribution of tables. On the HDFS DataNode found in the Hadoop cluster, Region Server operates. Region Server regions are in charge of handling, administering, and carrying out various tasks, including reading and writing HBase activities on that group of regions. 256 MB is a region's default size.

3. Zookeeper

It functions similarly to an HBase coordinator. It offers features including naming, distributed synchronization, server failure notice, and configuration information maintenance. Zookeeper is used by clients to connect to region servers.

5.3.3.4 Advantages

1. **Scalability:** HBase scales horizontally by distributing data across multiple servers (Region Servers) in a cluster.
2. **High Performance:** Provides real-time read/write access with low latency. Efficient for random read/write operations, making it suitable for transactional workloads.

3. **Flexible Schema Design:** Schema-less columns within column families, allowing dynamic addition of columns without modifying the schema. Supports sparse data storage, only storing non-null values which optimizes storage efficiency.
4. **Strong Consistency:** Ensures strong consistency for read and write operations, unlike many NoSQL databases that opt for eventual consistency.
5. **Fault Tolerance:** Built on top of HDFS (Hadoop Distributed File System), which provides reliable storage and automatic recovery from hardware failures.
6. **Automatic Sharding:** Automatically splits large tables into smaller regions and redistributes them across region servers for load balancing.
7. **Seamless Hadoop Integration:** Works well with Hadoop ecosystem tools such as MapReduce, Hive, and Spark for batch processing and data analysis.
8. **Data Versioning:** Stores multiple versions of a cell with different timestamps, enabling historical data retrieval.

5.3.3.5 Data Types

HBase doesn't have predefined data types like SQL databases. Instead, all data is stored as byte arrays (byte[]). Clients must convert data to and from byte arrays during read and write operations. The common data types of Hbase are shown in Table 5.3.18.

Table 5.3.18 Data types of Hbase

Data Type	Description	Conversion Method (Java)	Example
String	Textual data	Bytes. toBytes(String)	"Alice" → Bytes. toBytes("Alice")
Integer	32-bit integer	Bytes.toBytes(int)	12345 → Bytes. toBytes(12345)
Long	64-bit integer	Bytes.toBytes(long)	1234567890L → Bytes. toBytes(1234567890L)
Float	32-bit floating point	Bytes.toBytes(float)	12.34f → Bytes. toBytes(12.34f)
Double	64-bit floating point	Bytes. toBytes(double)	123.456 → Bytes. toBytes(123.456)
Boolean	Boolean value	Bytes. toBytes(boolean)	true → Bytes. toBytes(true)
Byte[]	Raw binary data (files, images)	Stored directly as byte[]	byte[] imgData

5.3.3.6 Operations

HBase supports several core operations for managing and manipulating data.

1. Table Management Operations

- ◆ **Create Table:** Defines a table with column families.

Example: `create 'Users', 'Info'`

- ◆ **List Tables:** Lists all available tables.

Example: `list`

- ◆ **Describe Table:** Displays table schema.

Example: `describe 'Users'`

- ◆ **Disable Table:** Disables a table before deletion or modification.

Example: `disable 'Users'`

- ◆ **Delete Table:** Deletes a table (must be disabled first).

Example: `drop 'Users'`

2. Data Manipulation Operations

- a. **Inserting Data (PUT):** Adds or updates a row in a table.

Example:

```
put 'Users', 'user001', 'Info:Name', 'Alice'
```

```
put 'Users', 'user001', 'Info:Age', '30'
```

- b. **Reading Data (GET):** Retrieves data for a specific row.

Example:

```
get 'Users', 'user001'
```

Returns all columns of the specified row unless a specific column is mentioned.

- c. **Deleting Data (DELETE):** Removes specific column data or entire row.

Example:

```
delete 'Users', 'user001', 'Info:Age'
```

To delete an entire row:

Example:

```
deleteall 'Users', 'user001'
```

5.3.4 Overview of Neo4j

The most well-known database management system is Neo4j, which is a NoSQL database system. Neo4j is unique among database management systems since it has characteristics of its own and is made to effectively store and query densely linked data, which sets it apart from MySQL or MongoDB. Unlike tables and rows, Neo4j utilizes a graph model composed of nodes, relationships, and properties, making it ideal for handling complex, interconnected data. Its schema-free nature offers flexibility, allowing dynamic data structures that evolve over time.

Neo4j is widely used in various applications such as social networks, fraud detection, recommendation systems, knowledge graphs, and network management. Its efficiency in managing relationships between entities makes it particularly suitable for scenarios where data interconnections are essential. The database offers ACID-compliance, scalability, high-performance read and write operations, and seamless integration with popular programming languages. Additionally, Neo4j provides tools for visualization and analytics, making it a preferred choice for enterprises seeking to leverage the full potential of their connected data.

5.3.4.1 Key Features

Key features of Neo4j Database include:

1. **Native Graph Storage and Processing:** Neo4j stores data as a graph directly rather than mapping it to relational structures, providing superior performance for graph traversal operations.
2. **ACID Compliance:** Ensures reliable transactions with strong consistency, making it suitable for enterprise-grade applications requiring data integrity.
3. **High Performance and Scalability:** Optimized for both read and write operations, Neo4j can handle billions of nodes and relationships with low latency, supporting horizontal scaling through clustering.
4. **Schema-Free Model:** Provides flexibility to adapt to changing requirements and evolving data structures without the need for predefined schemas.
5. **Index-Free Adjacency:** Relationships are stored as first-class entities, enabling fast, real-time traversal without relying on complex joins or indexing mechanisms.
6. **Built-in Data Visualization:** Includes tools for visualizing graphs, making it easier to explore and understand complex datasets.
7. **Comprehensive Security Features:** Offers role-based access control, user authentication, and fine-grained data access policies to protect sensitive information.
8. **APIs and Language Support:** Provides seamless integration with popular programming languages like Java, Python, JavaScript, and Go, along with a RESTful API.

5.3.4.2 Data Model

The data model for a Neo4j database is based on a Property Graph Model, which consists of four core elements.

1. **Nodes:** Fundamental data entities representing objects, concepts, or people.

Example: Person, Product, Location

2. **Relationships:** Directed connections between nodes representing how entities are associated.

Example: (Alice)-[:FRIEND]->(Bob) where FRIEND is the type of relationship.

3. **Properties:** Attributes or metadata attached to nodes and relationships.

4. **Labels:** Tags applied to nodes to categorize them.

5.3.4.3 Working of Neo4j

Neo4j does not save or display data in a tabular or JSON format; instead, it displays it as a graph. Nodes are used here to represent all of the data, and relationships between nodes may be established. This distinguishes it from other database management systems since it implies that the whole database collection will resemble a graph.

MS Access, SQL server, all the relational database management systems use tables to store or present the data with the help of columns and rows, but Neo4j doesn't use tables, rows or columns in the old-school style to store or present the data. A graph database uses graph theory to store, map, and query relationships. Figure 5.3.8 denoted the working structure of Neo4j. It consists of nodes, edges, and properties, where:

- ◆ **Nodes** represent entities such as people, businesses, or any data item.
- ◆ **Edges** (or relationships) connect nodes and illustrate how entities are related.
- ◆ **Properties** provide additional information about nodes and relationships.

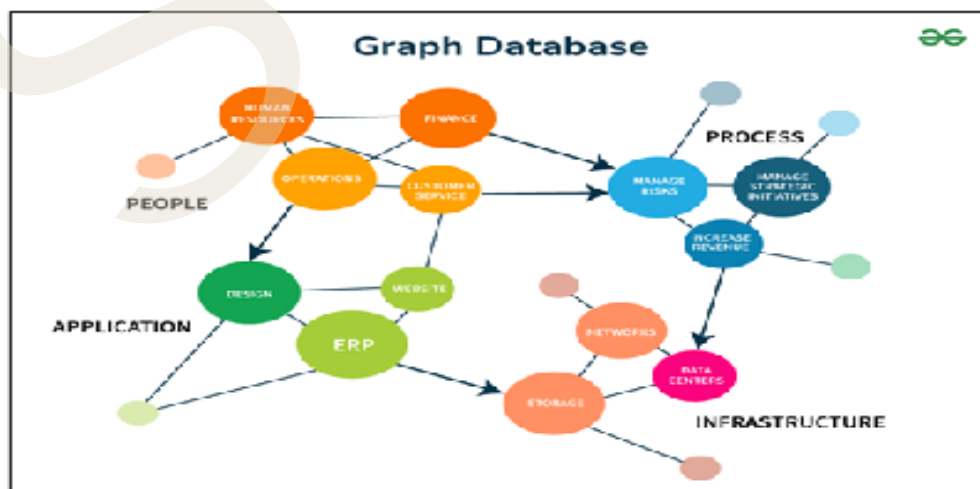


Fig. 5.3.8 Working structure of Neo4j

Compared to conventional relational databases, graph databases can more naturally and intuitively model real-world scenarios thanks to its structure.

5.3.4.4 Advantages

1. **High Performance for Connected Data:** Neo4j excels in handling complex, interconnected data by using native graph storage and index-free adjacency.
2. **Flexible Schema-Free Model:** No predefined schema is required, allowing for easy adaptation to evolving data structures.
3. **Intuitive Data Representation:** Uses a graph model (nodes, relationships, properties) that closely resembles how data is structured in the real world.
4. **Powerful Query Language (Cypher):** Simple, expressive, and declarative language designed specifically for graph traversal and pattern matching.
5. **Scalability & High Availability:** Supports clustering and replication, allowing horizontal scaling to handle increasing workloads.
6. **ACID Compliance:** Provides strong consistency and reliability of transactions, ensuring data integrity.
7. **Robust Security Features:** Supports role-based access control, authentication, and fine-grained security to protect sensitive data.
8. **Community & Enterprise Support:** Strong community support with comprehensive documentation and an enterprise edition offering additional features for production use.

5.3.4.5 Data Types

Neo4j, a leading graph database, supports a variety of data types that allow users to store and manage complex relationships and entities efficiently. These data types are essential for defining properties of nodes and relationships in a graph structure. The common data types used in Neo4j are described in Table 5.3.19.

Table 5.3.19 Data Types used in Neo4j

Data Type	Description	Example Code (Cypher)	Example Data
String	Textual data	CREATE (p:Person {name: "Alice", city: "NY"})	"Alice", "New York"
Integer	Whole numbers	CREATE (p:Person {name: "Bob", age: 28})	28, -100, 0
Float	Decimal numbers	CREATE (p:Product {name: "Laptop", price: 999.99})	999.99, -45.67

Boolean	True/False values	CREATE (p:Person {name: "Charlie", isActive: true})	true, false
List (Array)	Ordered collection of values	CREATE (p:Person {name: "David", hobbies: ["Reading", "Hiking"]})	["Reading", "Hiking"]
Date	Calendar date (YYYY-MM-DD)	CREATE (e:Event {name: "Conference", date: date("2025-03-14")})	2025-03-14

5.3.4.6 Operations

Neo4j uses the Cypher query language for performing various operations on the graph database. The most common operations are:

A. Data Creation (CREATE): Creating nodes and relationships.

// Creating Nodes

```
CREATE (a:Person {name: "Alice", age: 30})
```

```
CREATE (b:Person {name: "Bob", age: 25})
```

// Creating Relationships

```
MATCH (a:Person {name: "Alice"}), (b:Person {name: "Bob"})
```

```
CREATE (a)-[:FRIEND]->(b)
```

B. Data Reading (MATCH & RETURN): Retrieving nodes, relationships, and their properties.

// Fetch all Person nodes

```
MATCH (p:Person)
```

```
RETURN p
```

// Fetch specific relationships

```
MATCH (a:Person)-[r:FRIEND]->(b:Person)
```

```
RETURN a.name, b.name, r
```

C. Data Updating: Modifying or updating data.

// Updating properties of a node

```
MATCH (p:Person {name: "Alice"})
```

```
SET p.age = 31
```

// Adding a new property

```
SET p.city = "New York"
```

D. Data Deletion (DELETE, DETACH DELETE): Removing nodes and relationships.

// Deleting a relationship only

```
MATCH (a:Person)-[r:FRIEND]->(b:Person)
```

```
DELETE r
```

// Deleting a node and its relationships

```
MATCH (p:Person {name: "Alice"})
```

```
DETACH DELETE p
```

Pattern Matching in Neo4j

Pattern matching is one of the core features of Neo4j, and it is fully supported through its query language, Cypher. Pattern matching in Neo4j allows you to search for specific graph structures (nodes and relationships) based on defined patterns. It is similar to how SQL performs joins, but in a more intuitive way that reflects the natural connections in a graph.

Example:

```
MATCH (p:Person)-[:FRIEND_OF]->(f:Person)
```

```
WHERE p.name = 'Alice'
```

```
RETURN f.name
```

This query finds all people (f) who are connected to a person named Alice through the FRIEND_OF relationship.

Recap

- ◆ MongoDB is an open-source document-oriented database that is designed to store a large scale of data and allows us to work with that data efficiently.
- ◆ Key features of MongoDB are document-oriented storage, scalability, high performance, indexing, aggregation framework, replication, ACID transactions, Ad-hoc queries, File Storage (GridFS) and security.
- ◆ Data Model components of MongoDB are database, collection, document, field.
- ◆ MongoDB works in two layers: application layer, data layer.
- ◆ Advantages of MongoDB are schema flexibility, scalability and high performance, high availability with replica sets, rich query language, ease of use and development, security features, automatic scaling and load balancing, and aggregation framework.
- ◆ Data Types of MongoDB are string, integer, double, boolean, null, array, object (Embedded Document), Object Id, undefined, binary data, date
- ◆ Operations of MongoDB are create, read, update, delete.
- ◆ Apache Cassandra is a powerful open-source NoSQL database designed to handle large amounts of data across many servers with high availability and no single point of failure.
- ◆ Key Features of Cassandra are decentralized architecture, scalability, high availability, tunable consistency, high write performance, flexible data model, Cassandra Query Language (CQL), eventual consistency, multi-data center replication, fault tolerance and durability.
- ◆ Data Model of Cassandra contains keyspace, tables, rows, columns, primary key, clustering key
- ◆ Working of Cassandra specifies basic terminology, operations, storage engine
- ◆ Advantages of Cassandra are high availability and fault tolerance, scalability, fast write performance, flexible data model, distributed and decentralized, linear performance scaling, efficient data distribution, built for high availability applications.
- ◆ Datatypes of Cassandra support various data types broadly categorized into: Primitive Data Types, Collection Data Types, User-Defined Types (UDTs).
- ◆ Cassandra operations are mainly centered around table operations like create, insert, truncate, drop.
- ◆ HBase is an open-source, distributed, non-relational database built on top of the Hadoop Distributed File System (HDFS). It is designed to handle large

amounts of structured and semi-structured data, providing real-time read/write access.

- ◆ HBase features are scalability, column-oriented storage, automatic sharding, strong consistency, high availability, real-time read/write access, flexible data model, integrated with hadoop ecosystem.
- ◆ HBase data model components are table, row, column family and column qualifier.
- ◆ HBase working architecture has 3 main components: HMaster, Region Server, Zookeeper.
- ◆ Advantages of HBase are scalability, high performance, flexible schema design, strong consistency, fault tolerance, automatic sharding, seamless hadoop integration, data versioning.
- ◆ Data types of HBase include string, integer, long, float, double, boolean, byte[].
- ◆ HBase supports several core operations for managing and manipulating data: table management operations, data manipulation operations.
- ◆ Neo4j utilizes a graph model composed of nodes, relationships, and properties, making it ideal for handling complex, interconnected data. Its schema-free nature offers flexibility, allowing dynamic data structures that evolve over time.
- ◆ Key features of Neo4j database include native graph storage and processing, ACID compliance, high performance and scalability, schema-free model, index-free adjacency, built-in data visualization, comprehensive security features, apis and language support.
- ◆ The data model for a Neo4j database is based on a property graph model, which consists of four core elements: nodes, relationships, properties, labels.
- ◆ Working of Neo4j consists of nodes, edges, and properties.
- ◆ Advantages of Neo4j include high performance for connected data, flexible schema-free model, intuitive data representation, powerful query language (Cypher), scalability and high availability, ACID compliance, robust security features, community & enterprise support.
- ◆ Data Types of Neo4j are string, integer, float, boolean, list, date.
- ◆ Operations of Neo4j uses data creation (CREATE), data reading (MATCH & RETURN), data updating, data deletion (DELETE, DETACH DELETE).

Objective Type Questions

1. Which type of database is MongoDB?
2. Which feature of MongoDB provides various indexing options?
3. Document in MongoDB can be defined as.....
4. Which component of MongoDB is utilized for administrative tasks as well as inquiries and data modifications?
5. Name any one advantage of MongoDB.
6. Write an example of a string data type which is used in MongoDB.
7. `db.collection.find()` retrieves
8. Which type of database is Cassandra?
9. The tunable consistency feature of Cassandra defines.....
10. Define Keyspace in Cassandra.
11. Which component of Cassandra is denoted as a collection of nodes?
12. Name any one advantage of Cassandra.
13. Write an example of Collection Data Type which is used in Cassandra
14. Write the syntax to delete a table in Cassandra.
15. Which type of database is HBase?
16. The column-oriented storage feature of HBase defines.....
17. Define Column Qualifier in HBase.
18. Which component of HBase is denoted as the functions similar to an HBase coordinator?
19. Name any one advantage of HBase.
20. Write any one data type used in HBase.
21. Write the syntax to Create Table in HBase.
22. Which type of database is Neo4j?
23. The Schema-Free Model feature in Neo4j defines.....
24. The data model for a Neo4j database is based on a
25. Which component of Neo4j database represents entities such as people, businesses, or any data item?

26. Name any one advantage of Neo4j.
27. Which data type is used for denoting decimal numbers in Neo4j?
28. Example for Data Reading in Neo4j.

Answers to Objective Type Questions

1. MongoDB is an open-source document-oriented database categorized under the NoSQL database.
2. Indexing
3. A record in BSON format (similar to a row). Document is made of fields. It is similar to a tuple in RDBMS.
4. MongoDB Shell
5. Rich Query Language (It supports powerful, expressive queries for filtering, sorting, Aggregating or transforming data).
6. { "name": "Akash" }
7. It is used to retrieve documents from the collection.
8. Apache Cassandra is a powerful open-source NoSQL database designed to handle large amounts of data across many servers with high availability and no single point of failure.
9. Offers adjustable consistency levels (e.g., ONE, QUORUM, ALL) to balance between consistency, availability, and performance based on application needs.
10. The top-level namespace for tables and defines properties like replication strategy and replication factor.
11. Data Center
12. High Availability and Fault Tolerance
13. List example: ['Apple', 'Banana']
14. drop table Register; //Register is a table name.
15. HBase is an open-source, distributed, non-relational database built on top of the Hadoop Distributed File System (HDFS).
16. Data is stored in a column-family-oriented model, which provides efficient read/write operations for sparse data and allows flexible schema design.
17. The name of a specific column within a column family. Columns are dynamic, they can be created on the fly without modifying the table schema.

18. Zookeeper
19. Automatic Sharding
20. String
21. create 'Users', 'Info' //User is a table name.
22. Neo4j is a graphical database.
23. Provides flexibility to adapt to changing requirements and evolving data structures without the need for predefined schemas.
24. Property Graph Model
25. Nodes
26. Powerful Query Language (Cypher)
27. Float
28. // Fetch all Person nodes

MATCH (p:Person)

RETURN p

Assignments

1. Explain the working structure of MongoDB. Briefly define its data model and its operations.
2. Discuss the key components of Cassandra database. Describe the various data types and its operations.
3. What is HBase? Describe its major components and explain its advantages.
4. Describe Neo4j database. Discuss its working and data model in detail.

Suggested Reading

1. Redmond, E., & Wilson, J. R. (2018). *Seven databases in seven weeks: A guide to modern databases and the NoSQL movement* (2nd ed.). Pragmatic Bookshelf.

2. Wu, X. (B.), Kadambi, S., Kandhare, D., & Ploetz, A. (2018). *Seven NoSQL databases in a week: Get up and running with the fundamentals and functionalities of seven of the most popular NoSQL databases*. Packt Publishing.

Reference

1. Bradshaw, S., Brazil, E., & Chodorow, K. (2020). *MongoDB: The definitive guide* (3rd ed.). O'Reilly Media.
2. Carpenter, J., & Hewitt, E. (2022). *Cassandra: The definitive guide* (Rev. 3rd ed.). O'Reilly Media.
3. George, L. (2017). *HBase: The definitive guide* (2nd ed.). O'Reilly Media.
4. Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.

Unit 4

NoSQL in Practice

Learning Outcomes

The learners will be able to:

- ◆ identify common applications of NoSQL databases in real-world scenarios such as big data, e-commerce, and social media
- ◆ recognize the challenges of implementing NoSQL, such as consistency issues, query complexity, and security concerns
- ◆ explain different NoSQL deployment strategies, including cloud based, on-premises, and hybrid approaches
- ◆ summarize performance optimization techniques and scalability strategies used in NoSQL databases

Prerequisites

Did you play online multiplayer games?

In online multiplayer games like Fortnite, PUBG, or Call of Duty, millions of players engage in real-time battles, requiring fast, scalable, and low latency data storage. NoSQL databases play a crucial role in managing player rankings, game states, matchmaking, and chat systems. These databases ensure smooth gameplay by handling high volumes of read and write operations efficiently.

NoSQL applications offer several advantages, making them ideal for modern data-driven environments. They provide high scalability by allowing horizontal scaling, enabling businesses to handle growing data volumes efficiently. Their schema-less design offers flexibility, allowing dynamic changes without predefined structures. NoSQL databases ensure high availability and fault tolerance through built-in replication and distributed architecture. They also deliver low-latency performance, making them suitable for real-time applications such as gaming, IoT, and streaming. Additionally, NoSQL supports varied data models (key-value, document, column-family, and graph), catering to diverse use cases like e-commerce, social media, and big data analytics.

Keywords

Redis, Neo4j, MongoDB, InfluxDB, Apache HBase



Discussion

NoSQL ("Not Only SQL") databases are designed to handle unstructured, semi structured, and structured data efficiently. They offer flexibility, scalability, and high performance for modern applications.

NoSQL databases have several key characteristics that make them suitable for modern applications. They feature a schema-less design, allowing dynamic changes without predefined structures. Horizontal scalability enables them to scale out by adding more servers, while their distributed architecture ensures data is spread across multiple nodes for improved reliability. NoSQL supports flexible data models, including key value, document, column family, and graph based storage, providing versatility for different use cases. Additionally, they offer high availability through built-in replication and fault tolerance. Unlike traditional relational databases that follow strict ACID compliance, many NoSQL databases adopt eventual consistency, using the BASE model (Basically Available, Soft state, Eventually consistent) to balance availability and performance.

5.4.1 Applications of NoSQL

NoSQL databases are widely used in various domains due to their flexibility, scalability, and high performance capabilities. Some key applications include:

5.4.1.1 NoSQL Databases in Big Data and Analytics

NoSQL databases play a crucial role in data warehousing by acting as data lakes or large scale storage systems that can ingest, store, and manage vast amounts of structured, semi structured, and unstructured data. Unlike traditional relational databases, which rely on a fixed schema and structured tables, NoSQL databases provide schema flexibility, making them ideal for storing diverse data formats from various sources.

NoSQL handles massive datasets from multiple sources. Organizations collect data from numerous sources, such as:

- ◆ Transactional Systems – E-commerce platforms, banking transactions, and enterprise applications
- ◆ Social Media & Web Analytics – Clickstream data, social media feeds, and customer interactions
- ◆ IoT Devices & Sensors – Logs from industrial machines, healthcare wearables, and smart devices
- ◆ Enterprise Applications – CRM, ERP, and business intelligence tools

NoSQL databases aggregate and store these datasets in a way that allows scalability and efficient retrieval without the performance bottlenecks associated with traditional relational databases.

Advantages of NoSQL Databases in Data Warehousing

NoSQL databases offer scalability through horizontal scaling, enabling businesses to distribute workloads across multiple servers, which is especially advantageous in

big data environments where data volumes grow exponentially. Their faster query performance is achieved through distributed storage and parallel processing, allowing for efficient data retrieval, analytics, and reporting. Additionally, NoSQL databases provide a flexible schema, unlike relational databases that require predefined structures, making it easier to adapt to evolving business needs. Furthermore, they ensure high availability and fault tolerance by offering automatic replication across multiple nodes, guaranteeing data availability even in the event of system failures.

Examples of NoSQL Databases Used in Data Warehousing

- ◆ **MongoDB:** A document oriented NoSQL database that supports JSON like data storage. It is widely used for real time analytics and business intelligence due to its ability to handle large volumes of semi structured and unstructured data.
- ◆ **Apache HBase:** A column family NoSQL database built on Hadoop, optimized for real-time analytics on massive datasets. It is widely used in big data applications, such as processing web-scale clickstream data and customer behavior analytics.

5.4.1.2 NoSQL Databases in Real-Time Analytics

In today's digital era, businesses depend on real-time analytics for decision-making, monitoring, and reporting. Traditional relational databases often struggle with high-velocity data due to their rigid structure and complex queries, whereas NoSQL databases, especially columnar and key-value stores, are designed for rapid reads and writes, making them well-suited for such applications. Real time analytics enables swift responses to changing conditions like fraud detection, stock market fluctuations, and website personalization. It also supports operational monitoring in industries such as e-commerce, banking, healthcare, and IT by tracking system performance and detecting anomalies. Moreover, by analyzing user behavior, it enhances customer experience through personalized recommendations and optimized marketing strategies, driving engagement and satisfaction.

How NoSQL Databases Power Real-Time Analytics?

NoSQL databases power real time analytics by offering low latency reads and writes, enabling fast data ingestion and retrieval for quick query execution. Their horizontal scalability ensures workloads are distributed across multiple nodes, preventing bottlenecks even in high-traffic scenarios. Additionally, NoSQL databases support distributed data processing, allowing seamless data replication and parallel execution across multiple servers. With their schema flexibility, they can efficiently handle structured, semi-structured, and unstructured data, making them highly adaptable to dynamic analytics requirements.

Industry Use Cases of NoSQL in Real-Time Analytics

NoSQL databases play a crucial role in real-time analytics across various industries. In e-commerce, they power recommendation engines by analyzing customer browsing behavior, purchase history, and preferences, with Amazon DynamoDB enabling

personalized shopping experiences. Streaming services utilize NoSQL databases to analyze real-time customer interactions, as seen in Netflix's use of Apache Cassandra for content recommendations and performance monitoring. In financial services, NoSQL databases help detect fraudulent transactions by analyzing spending patterns and anomalies, with Aerospike being a preferred choice for fraud prevention within milliseconds. In healthcare, real-time patient monitoring relies on NoSQL databases to process vital signs, wearable device data, and medical records, aiding early diagnosis, as demonstrated by Couchbase-powered healthcare analytics platforms. Similarly, cybersecurity systems leverage NoSQL databases for threat detection, analyzing network logs, login patterns, and access behavior, with Elasticsearch widely used for real time log analytics and intrusion detection.

5.4.1.3 NoSQL Database in Machine Learning and AI Applications

Machine learning and AI applications depend on massive datasets for training and inference, requiring efficient data storage and retrieval mechanisms. NoSQL databases excel in handling these large-scale datasets by offering high speed data access, scalability, and flexible data structures. They are particularly useful for storing vector embeddings, graph-based relationships, and unstructured data, which are essential for AI driven applications.

For instance, graph based NoSQL databases like Neo4j are widely used in fraud detection and recommendation systems. In fraud detection, they analyze complex relationships between entities, such as transactions, users, and locations, to identify suspicious patterns. Similarly, in recommendation engines, graph-based NoSQL databases help model user preferences and product interactions, enhancing personalized recommendations.

Additionally, document oriented NoSQL databases like MongoDB and key value stores like Redis are leveraged in AI applications for storing training datasets, caching real time predictions, and managing unstructured text or image data. By providing scalable, high performance data access, NoSQL databases significantly enhance the efficiency and effectiveness of machine learning and AI-driven systems.

5.4.1.4 NoSQL Applications in Social Media and Content Management

Social media platforms and content management systems generate large volumes of dynamic, user-generated content, including posts, comments, likes, shares, images, and videos. NoSQL databases are ideal for handling this high velocity, high variety data due to their scalability, flexibility, and fast data processing capabilities.

For instance, document oriented databases like MongoDB efficiently store JSON-like data structures, making them suitable for managing social media posts, comments, and multimedia content. Key value stores such as Redis enable fast caching and real-time updates, improving user experience by reducing latency in feed updates and notifications.

Additionally, wide-column databases like Apache Cassandra are widely used for distributed data storage, ensuring seamless content availability across global data centers.

Social media giants like Facebook, Twitter, and Instagram leverage NoSQL databases to store, retrieve, and analyze massive datasets while maintaining high availability and low latency. These databases also support real-time analytics, recommendation systems, and trend analysis, enhancing user engagement and personalization.

5.4.1.5 NoSQL Applications in E-Commerce

E-commerce platforms handle vast amounts of dynamic data, including product catalogs, user sessions, transactions, and customer interactions. NoSQL databases are widely used in e-commerce due to their scalability, fast data retrieval, and schema flexibility, enabling seamless shopping experiences for users.

For product catalog management, document-oriented databases like MongoDB efficiently store complex, hierarchical product information, including descriptions, images, pricing, and inventory details. Their flexible schema allows businesses to easily update and expand product attributes without requiring database restructuring.

For managing user sessions, key value stores such as Redis provide real time session storage and caching, ensuring smooth user experiences by maintaining login states, shopping carts, and browsing history. These databases enable fast retrieval of user preferences and recently viewed items, enhancing personalization and recommendation systems.

Additionally, wide-column stores like Apache Cassandra support high-volume transaction processing, ensuring reliability and scalability for handling millions of concurrent users. Leading e-commerce platforms like Amazon, eBay, and Flipkart leverage NoSQL databases to provide personalized recommendations, optimize inventory management, and enhance overall site performance.

5.4.1.6 NoSQL Applications in Healthcare and Genomics

NoSQL databases play a crucial role in healthcare and genomics by enabling the storage, retrieval, and analysis of massive, complex datasets such as electronic health records (EHRs), medical imaging, patient monitoring data, and genomic sequences. Due to their scalability, flexibility, and high-performance processing, NoSQL databases help improve patient care, medical research, and precision medicine.

In electronic health records (EHRs), document oriented databases like MongoDB efficiently store semi structured patient data, including medical histories, prescriptions, and diagnostic reports, allowing real time access for healthcare providers. For real-time patient monitoring, key-value stores like Redis are used to store sensor-generated data from wearable devices, ensuring quick data retrieval for remote patient monitoring and predictive analytics.

In genomics, the analysis of DNA sequences requires handling petabytes of unstructured and semi-structured data. Wide-column stores such as Apache Cassandra enable high-throughput storage and distributed processing of genomic data, facilitating large scale computational tasks in gene mapping and mutation detection. Similarly, graph databases like Neo4j help in modeling biological relationships, such as protein interactions and disease pathways, aiding in drug discovery and personalized medicine.

With NoSQL databases, healthcare institutions and research centers can efficiently manage big data challenges, improving diagnostic accuracy, treatment effectiveness, and biomedical research advancements.

5.4.1.7 NoSQL Applications in IoT and Sensor Data

The Internet of Things (IoT) generates massive volumes of time series and streaming data from connected devices, including smart sensors, industrial machines, wearables, and autonomous vehicles. NoSQL databases are well suited for handling this high velocity, real time data due to their scalability, fast ingestion, and flexible schema.

For storing and analyzing time series data, time-series databases like InfluxDB and wide-column stores like Apache Cassandra are commonly used. These databases efficiently handle timestamped data, allowing businesses to track sensor readings, environmental conditions, and machine logs over time.

For real-time streaming data, key value stores such as Redis and document-oriented databases like MongoDB enable low latency processing, making them ideal for applications like predictive maintenance, smart city monitoring, and real-time anomaly detection. Additionally, graph databases like Neo4j help analyze device relationships and network structures, supporting applications in cybersecurity, logistics, and fleet management.

By leveraging NoSQL databases, IoT ecosystems can efficiently store, process, and analyze vast amounts of sensor generated data, enabling real time insights, automation, and smarter decision-making across various industries.

5.4.2 Challenges in NoSQL Implementation

NoSQL databases have gained significant traction due to their scalability, flexibility, and ability to handle large volumes of unstructured data. However, their implementation poses several challenges that organizations must address to ensure successful adoption. Below are some of the key challenges encountered while implementing NoSQL databases:

5.4.2.1 Querying and Indexing Challenges

NoSQL databases often have limited query capabilities compared to SQL databases, which offer powerful query languages and indexing mechanisms. As a result, developers may need to design custom indexing strategies to enhance performance, adding complexity to application development.

5.4.2.2 Data Consistency

NoSQL databases often prioritize availability over consistency, as described by the CAP theorem. This means that applications using NoSQL databases must manage eventual consistency, where data updates may not be immediately reflected across all nodes.

5.4.2.3 Limited ACID Compliance (Transaction Support)

Many NoSQL databases prioritize performance and scalability over full ACID (Atomicity, Consistency, Isolation, Durability) compliance, which can result in data consistency challenges. To address this, developers often need to implement mechanisms like eventual consistency, adding complexity to data management.

5.4.2.4 Security Concerns

Designed primarily for scalability, many NoSQL databases may compromise on security, leading to potential vulnerabilities. Weak authentication, lack of role based access controls, and limited encryption support can increase the risk of exposing sensitive data.

5.4.2.5 Complex Data Modeling

NoSQL databases require a different and often complex approach to data modeling, which can be challenging for developers familiar with relational databases. Selecting the appropriate data model, whether document, key-value, column family, or graph demands a thorough understanding of the application's specific requirements.

5.4.2.6 Lack of Standardization

NoSQL databases lack the standardization found in SQL databases, which follow a unified query language like SQL. With significant variations in design, architecture, and querying mechanisms, transitioning between different NoSQL databases can be challenging for developers and complicates seamless integration.

5.4.2.7 Migration Complexity

Migrating from traditional SQL databases to NoSQL requires substantial effort in data transformation, application rewrites, and adapting to new consistency models. Organizations must invest in the right tools and expertise to ensure a smooth transition while maintaining data integrity and preventing loss.

5.4.2.8 Tooling and Ecosystem Maturity

NoSQL databases often lack the mature ecosystem and extensive tooling available for SQL databases, which provide robust management, monitoring, and analytics solutions. As a result, debugging and performance tuning in NoSQL environments can be challenging due to limited support for comprehensive profiling tools.

5.4.2.9 Interoperability Issues

Many NoSQL databases use proprietary formats and APIs, which can make integration with existing enterprise systems challenging. To enable seamless data exchange between NoSQL and other database systems, organizations often need to develop custom middleware solutions.

5.4.2.10 Cost Considerations

Although NoSQL databases are often open-source, the infrastructure and operational costs can be substantial due to their distributed architectures and scaling demands. Additionally, managing a NoSQL deployment requires skilled personnel, further increasing overall expenses.

5.4.3 Deployment Strategies for NoSQL Databases

Deploying NoSQL databases requires careful planning to ensure scalability, performance, and reliability. Different strategies are used based on application needs, data distribution, and infrastructure constraints. Below are key deployment strategies for NoSQL databases:

5.4.3.1 Single Node Deployment

A single-node deployment is ideal for small-scale applications, development, and testing environments, as it runs the entire database instance on a single machine. While this setup is simple and easy to manage, it lacks scalability and fault tolerance, making it unsuitable for large or high-availability applications.

5.4.3.2 Cluster-Based Deployment

A cluster-based deployment uses multiple nodes to distribute data and processing workloads, making it suitable for large-scale applications. This approach enhances scalability, ensures high availability, and enables load balancing. However, it requires careful configuration for replication and data partitioning to maintain consistency and performance.

5.4.3.3 Sharding (Horizontal Partitioning)

Sharding, or horizontal partitioning, divides large datasets across multiple nodes using a partitioning key, allowing queries and writes to be distributed efficiently across multiple servers. This improves performance and scalability by preventing any single node from becoming a bottleneck. However, a well planned sharding strategy is essential to avoid data hotspots and ensure balanced workloads.

5.4.3.4 Replication

Replication involves copying data across multiple nodes to enhance availability and fault tolerance. It can be implemented using master slave (primary-secondary) or master-master replication models. This approach supports disaster recovery and improves read scalability, but it requires proper synchronization management to maintain data consistency across nodes.

Backup Strategies and Disaster Recovery in Replication Systems:
Replication-based systems focus on maintaining real-time or near-real-time copies of data across multiple servers or locations to ensure high availability and quick recovery. In these systems, data from the primary server is duplicated to one or more secondary servers using synchronous or asynchronous replication. If the primary system fails, a

replica can take over with minimal downtime and data loss. Disaster recovery in such environments involves configuring failover mechanisms, monitoring replication health, and regularly testing the system to ensure that data remains consistent and available. This approach is especially useful for mission-critical applications where continuous access to up-to-date information is essential.

5.4.3.5 Hybrid Cloud Deployment

A hybrid deployment combines on premises and cloud based NoSQL database instances, allowing organizations to balance cost, security, and scalability. This approach is particularly useful for disaster recovery and efficiently managing peak loads by leveraging cloud resources when needed.

Backup Strategies and Disaster Recovery in Hybrid Systems:

Hybrid systems combine on-premises and cloud infrastructure to create a flexible and resilient backup and disaster recovery approach. In this setup, data is regularly backed up to local storage for quick recovery and simultaneously replicated or backed up to the cloud for off-site protection. This dual strategy ensures data is safe even if one layer fails, such as in the case of a local hardware crash or a natural disaster affecting the data center. Disaster recovery in hybrid systems involves predefined recovery plans that switch operations to the cloud environment when on-site resources are compromised, maintaining business continuity with acceptable Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO).

5.4.3.6 Multi-Cloud Deployment

A multi-cloud deployment distributes the database across multiple cloud providers to prevent vendor lock-in and enhance availability. This approach increases resilience against cloud provider failures but requires efficient data synchronization and latency management to ensure seamless performance.

5.4.3.7 Containerized Deployment (Docker & Kubernetes)

A containerized deployment utilizes technologies like Docker and orchestration tools such as Kubernetes to streamline database deployment and management. This approach offers portability, automated scaling, and simplified deployment across different environments. However, it requires proper volume management to ensure persistent storage and data integrity.

5.4.3.8 Serverless NoSQL Deployment

A serverless NoSQL deployment relies on cloud-based databases where the provider manages infrastructure, scaling, and provisioning. This eliminates the need for manual resource management, allowing applications to scale automatically based on demand. Examples of serverless NoSQL databases include AWS DynamoDB, Azure Cosmos DB, and Google Firestore.

5.4.4 Performance and Scalability of NoSQL Databases

The performance of NoSQL databases depends on their architecture, data model, and deployment strategy. Unlike traditional SQL databases, NoSQL databases are optimized for scalability, high-speed read and write operations, and handling large volumes of unstructured or semi-structured data. Features like horizontal scaling, sharding, and replication enhance performance by distributing workloads across multiple nodes. However, factors such as data consistency models, indexing strategies, and query optimization can impact speed and efficiency. While NoSQL databases excel in handling big data and real-time applications, their performance may vary based on use cases, requiring careful tuning and configuration.

5.4.4.1 Performance Optimization

Optimizing performance in NoSQL databases involves several key strategies to enhance efficiency and scalability.

- ◆ **Schema Flexibility:** NoSQL databases support dynamic schema updates, allowing applications to evolve without strict data structure constraints. This flexibility reduces overhead and improves read and write performance by eliminating the need for costly schema migrations.
- ◆ **Indexing Strategies:** Proper indexing enhances query performance by enabling faster data retrieval. However, excessive indexing can slow down write operations, as each data insertion or update must also modify the indexes. Finding the right balance between query speed and write efficiency is crucial.
- ◆ **Data Partitioning:** Sharding distributes data across multiple nodes, preventing single-node bottlenecks and enabling parallel processing. A well designed partitioning strategy ensures balanced workloads and improves overall system performance.
- ◆ **In-Memory Caching:** Using caching layers like Redis or Memcached reduces the number of direct database queries by storing frequently accessed data in memory. This approach significantly improves response times and reduces database load.
- ◆ **Efficient Query Execution:** Optimizing query patterns, avoiding full-table scans, and leveraging appropriate aggregation techniques help improve retrieval speed. By structuring queries efficiently, developers can minimize computational overhead and enhance database performance.

Performance optimization in databases involves techniques that improve the speed and efficiency of data retrieval and management. One common issue is slow query performance, especially when dealing with large datasets. For example, consider a query like `'SELECT * FROM employees WHERE department = 'Sales';'` on a table with millions of records and no index on the `'department'` column. Without an index, the database must perform a full table scan, checking each row to find matches, which is time-consuming. By creating an index on the `'department'` column using `'CREATE INDEX idx_dept ON employees(department);'`, the database can quickly locate all

'Sales' department entries using the index, significantly reducing query execution time. This simple optimization can transform a multi-second query into a sub-second one, demonstrating how indexing enhances performance.

5.4.4.2 Factors Affecting NoSQL Performance

Several factors influence the performance of NoSQL databases, requiring careful consideration for optimization.

- ◆ **Workload Type:** Read-heavy or write-heavy workloads require different optimization strategies.
- ◆ **Replication Lag:** Data synchronization delays in distributed environments may impact real-time performance.
- ◆ **Storage Mechanisms:** The choice of disk-based or in-memory storage affects data retrieval speed.
- ◆ **Network Latency:** Distributed databases introduce network overhead, impacting response times.

5.4.4.3 Scalability Techniques in NoSQL Databases

NoSQL databases are designed for scalability, allowing them to handle large amounts of data and traffic efficiently. They achieve this through various techniques that optimize resource utilization and ensure high availability.

Horizontal Scaling (Scale-Out): NoSQL databases primarily scale horizontally by adding more servers (nodes) to distribute the workload. This improves performance, enhances fault tolerance, and prevents a single point of failure.

Vertical Scaling (Scale-Up): Some NoSQL databases also support vertical scaling, where the capacity of a single server is increased by adding more CPU, RAM, or storage. While this can improve performance, it has limitations and is less flexible than horizontal scaling.

Replication: To enhance availability and fault tolerance, NoSQL databases maintain copies of data across multiple nodes. Replication improves read performance by allowing multiple nodes to serve read requests simultaneously.

Load Balancing: Efficient distribution of requests across nodes prevents overloading any single node, ensuring optimal resource utilization and system stability. Load balancing is critical for maintaining consistent performance in large-scale deployments.

Sharding (Partitioning): Data is divided into smaller, more manageable chunks and distributed across multiple servers. This prevents bottlenecks, improves query performance, and enhances scalability by enabling parallel processing.

Auto-Scaling: Cloud-based NoSQL solutions, such as Amazon DynamoDB and Google Firestore, dynamically adjust resources based on demand. This ensures efficient scaling without manual intervention, optimizing cost and performance.

Eventual Consistency vs. Strong Consistency: Some NoSQL databases prioritize scalability by adopting eventual consistency, where data updates propagate over time. Others offer strong consistency to ensure immediate data accuracy, often at the cost of performance.

Optimizing NoSQL databases requires a combination of effective data modeling, resource management, and query optimization techniques. Proper data modeling is essential for improving query efficiency and storage utilization, ensuring that data is structured in a way that minimizes retrieval time. Implementing caching mechanisms, such as Redis or Memcached, helps reduce database load by storing frequently accessed data in memory, improving response times.

Balancing replication and sharding is crucial for maintaining both fault tolerance and performance. Replication ensures data availability, while sharding distributes data across multiple nodes to prevent bottlenecks. Additionally, optimizing query patterns by avoiding unnecessary computations and full scans enhances database efficiency. Finally, selecting the right NoSQL database type, whether key-value, document, column-family, or graph, based on the application's requirements ensures the best balance between scalability, speed, and flexibility.

5.4.4.5 Monitoring and Maintenance

Effective monitoring and maintenance are essential for ensuring the reliability and performance of NoSQL databases. Tracking key performance metrics such as latency, throughput, and error rates helps identify potential issues and optimize database operations. Regular monitoring allows for proactive troubleshooting, preventing performance degradation and downtime.

Automated backups play a critical role in maintaining data durability and recovery. By implementing scheduled backups, organizations can safeguard against data loss due to failures or corruption, ensuring business continuity.

For distributed NoSQL deployments, efficient cluster management is necessary to maintain stability and scalability. Tools like Kubernetes, ZooKeeper, and cloud-native orchestration solutions help automate resource allocation, replication, and failover processes, making it easier to manage large-scale NoSQL infrastructures.

5.4.4.6 Best Practices for Enhancing NoSQL Performance and Scalability

To maximize the efficiency and scalability of NoSQL databases, adopting best practices in data modeling, caching, and query optimization is essential. Proper data modeling ensures that queries are executed efficiently and storage is utilized optimally, reducing unnecessary overhead. Implementing caching mechanisms, such as Redis or Memcached, minimizes database load by storing frequently accessed data in memory, significantly improving response times.

Balancing replication and sharding is key to achieving both fault tolerance and performance. Replication enhances data availability, while sharding distributes data across multiple nodes to prevent bottlenecks and improve scalability. Optimizing query

patterns by avoiding unnecessary computations and full scans further enhances database performance.

Finally, selecting the right type of NoSQL database, whether key-value, document, column-family, or graph, based on the application's specific needs ensures the best balance between flexibility, speed, and scalability. By following these best practices, organizations can build efficient, high-performing NoSQL solutions.

Recap

- ◆ Applications of NoSQL ARE NoSQL Databases in Big Data and Analytics, NoSQL Databases in Real-Time Analytics, NoSQL Database in Machine Learning and AI Applications, NoSQL Applications in Social Media and Content Management, NoSQL Applications in E-Commerce, NoSQL Applications in Healthcare and Genomics, NoSQL Applications in IoT and Sensor Data.
- ◆ Challenges in NoSQL Implementation are Querying and Indexing Challenges, Data Consistency, Limited ACID Compliance (Transaction Support), Security Concerns, Complex Data Modeling, Lack of Standardization, Migration Complexity, Tooling and Ecosystem Maturity, Interoperability Issues, Cost Considerations
- ◆ Deployment Strategies for NoSQL Databases are Single Node Deployment, Cluster-Based Deployment, Sharding (Horizontal Partitioning), Replication, Hybrid Cloud Deployment, Multi-Cloud Deployment, Containerized Deployment (Docker & Kubernetes), Serverless NoSQL Deployment
- ◆ Performance and Scalability of NoSQL Databases: Factors Affecting NoSQL Performance
- ◆ Best Practices for Enhancing NoSQL Performance and Scalability
- ◆ Monitoring and Maintenance

Objective Type Questions

1. Which type of NoSQL database is best suited for handling time-series data in IoT applications?
2. Which NoSQL database is widely used for storing timestamped IoT data?
3. Which NoSQL database is commonly used for real-time streaming data processing in IoT?
4. Why are NoSQL databases preferred for IoT applications?

5. Which NoSQL database is suitable for analyzing relationships in IoT networks?
6. What is the main advantage of using Apache Cassandra for IoT data storage?
7. Which NoSQL database is widely used for caching real-time IoT data?
8. Why is schema flexibility important for IoT databases?
9. Which NoSQL database is best suited for fleet management and logistics in IoT?

Answers to Objective Type Questions

1. Time-Series Database
2. InfluxDB
3. Redis
4. They offer flexible schema and scalability
5. Neo4j
6. It supports distributed, high-throughput storage
7. Redis
8. IoT data structures change frequently
9. Neo4j

Assignments

1. Explain the role of NoSQL databases in handling IoT-generated data. How do they differ from traditional relational databases in terms of scalability and performance?
2. Discuss the advantages of using time-series databases like InfluxDB for storing IoT sensor data. Provide an example of an industry where time-series data is crucial.
3. Compare and contrast the use of Apache Cassandra and MongoDB for managing IoT data. Which database would be more suitable for a smart city project, and why?

4. Describe how NoSQL databases enable real-time streaming data processing in IoT applications. What role does Redis play in caching IoT sensor data?
5. What are the key challenges in managing IoT data, and how do NoSQL databases address these challenges? Discuss with relevant examples.
6. Explain the importance of schema flexibility in IoT applications. How do document-oriented NoSQL databases help in storing dynamic sensor data?
7. Discuss the role of graph databases like Neo4j in IoT applications. How can they be used for analyzing relationships between connected devices in a smart grid?
8. Design a data model using a NoSQL database for a real-time patient monitoring system in healthcare. Which NoSQL database would you choose and why?
9. Explain how predictive maintenance in industrial IoT (IIoT) benefits from NoSQL databases. Provide a case study or real-world example.
10. Evaluate the future of NoSQL databases in IoT applications. What emerging trends or technologies could further enhance their efficiency and scalability?

Suggested Reading

1. Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.
2. da Silva, M. D., & Tavares, H. L. (2015). *Redis essentials* (1st ed.). Packt Publishing.
3. Ostrovsky, D., Rodenski, Y., & Haji, M. (2014). *Pro Couchbase Server* (1st ed.). Apress.

Reference

1. Carpenter, J., & Hewitt, E. (2020). *Cassandra: The definitive guide* (2nd ed.). O'Reilly Media.
2. da Silva, M. D., & Tavares, H. L. (2015). *Redis essentials* (1st ed.). Packt Publishing.
3. George, L. (2017). *HBase: The definitive guide* (2nd ed.). O'Reilly Media.
4. Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems* (1st ed.). O'Reilly Media.

5. McCreary, D. G., & Kelly, A. M. (2013). *Making sense of NoSQL: A guide for managers and the rest of us* (1st ed.). Manning Publications.
6. Membrey, P., Hows, D., & Plugge, E. (2017). *The definitive guide to MongoDB: A complete guide to dealing with big data using MongoDB* (3rd ed.). Apress.
7. Ostrovsky, D., Rodenski, Y., & Haji, M. (2014). *Pro Couchbase Server* (1st ed.). Apress.
8. Redmond, E., & Wilson, J. R. (2018). *Seven databases in seven weeks: A guide to modern databases and the NoSQL movement* (2nd ed.). Pragmatic Bookshelf.
9. Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.
10. Sadalage, P. J., & Fowler, M. (2012). *NoSQL distilled: A brief guide to the emerging world of polyglot persistence* (1st ed.). Addison-Wesley.



Distributed Database and Future trends

Unit 1

Distributed Database Concepts

Learning Outcomes

Upon the completion of this unit, the learners will be able to:

- ◆ define key concepts related to Distributed Database Systems
- ◆ identify different architectures used in Parallel Database Systems
- ◆ explain the process of query processing in distributed database environments
- ◆ describe the concept of inter-query and intra-query parallelism in parallel databases

Prerequisites

Imagine a global online shopping company like Amazon that operates in multiple countries and serves millions of customers every day. Customers from different parts of the world place orders, track shipments, and browse products simultaneously. If all of this information were stored and processed in a single central system located in one country, it would create significant delays in data access, especially for users located far from that central location. During high-demand periods like festive sales or special promotions, the system could become overloaded, leading to slow responses, failed transactions, or even system crashes. Moreover, if the central system experiences a technical failure, the entire business operation could come to a halt, affecting customer satisfaction and revenue. Such scenarios highlight the need for a system that can handle large volumes of data, support high-speed access from various locations, ensure continuous availability, and offer scalability as business demands grow. This growing demand for performance, reliability, and speed in real-time environments makes it essential to rethink how data is stored, accessed, and managed in modern applications. Here comes the need for distributed databases.

Keywords

Data Distribution, Replication, Scalability, Query Processing, Fault Tolerance, Data Consistency, Parallel Processing, Communication Cost



Discussion

A database is a collection of information that is stored and managed using a computer system. In many cases, all the data is kept in a single location, known as a centralized database. However, as organizations grow and expand across different locations, storing all the data in one place may not be practical. This is where distributed databases come in. A distributed database is a database that is spread across multiple computers or locations. The computers that store and manage the database are connected through a network, and users can access the database from different locations. This setup makes data storage more efficient, reliable, and accessible.

6.1.1 Key Features of Distributed Databases

6.1.1.1 Data Distribution

Data distribution refers to the method of storing data across multiple physical or geographical locations rather than confining it to a single central server. This approach is commonly used in distributed database systems where data is fragmented and stored in different locations, either for performance, organizational needs, or redundancy. This setup not only enhances data accessibility but also reduces latency for users located in different regions, as they can access data from the nearest server node. It also supports load balancing, as the demand is spread across multiple servers rather than stressing a single point.

6.1.1.2 Replication

Replication is the process of copying and maintaining identical sets of data at more than one location. It is a key technique used to enhance both data availability and performance. If one location becomes temporarily unavailable due to network issues or system failures, other replicated sites can continue to serve data without interruption. Replication can be synchronous or asynchronous, depending on whether updates are immediately propagated or delayed. It also helps in read scalability, allowing multiple users to access copies of the same data from different sites simultaneously.

6.1.1.3 Independence

Data independence in a distributed system means that even though the data is physically spread across various locations, users experience it as if it were centralized. The complexity of data distribution is hidden from end users and application developers, who interact with the system through a single unified interface. This logical transparency is critical for user convenience and application portability. It allows for simplified querying and maintenance, as users don't need to know where data resides physically.

6.1.1.4 Reliability

Reliability in distributed database systems is achieved through redundancy and fault tolerance mechanisms. When one database server or site fails due to hardware malfunction or network issues, the system automatically redirects requests to other available sites or replicated data nodes. This ensures uninterrupted service and minimal

data loss, significantly increasing the system's robustness. High reliability is especially crucial for mission-critical applications where downtime or data unavailability can result in severe consequences.

6.1.1.5 Scalability

Scalability refers to the system's ability to expand seamlessly as data volume and user demand grow. In a distributed architecture, new nodes or locations can be added without affecting the existing infrastructure or disrupting the ongoing operations. This modular growth model allows organizations to scale horizontally by simply incorporating additional hardware or cloud-based nodes. Scalability ensures that the system remains responsive and efficient even as workloads increase, making it suitable for dynamic environments with evolving data and user requirements.

6.1.2 Advantages of Distributed Databases

6.1.2.1 Faster Access

Distributed database systems enable quicker data retrieval by allowing users to access information from the server closest to their geographical location. This reduces the time taken for data to travel across networks, significantly minimizing latency and improving system responsiveness. As a result, users experience faster query execution and smoother interactions with the system, which is especially beneficial in applications that require real-time data processing or high-speed transactions.

6.1.2.2 Improved Reliability

One of the key advantages of a distributed database is its enhanced system reliability and fault tolerance. If a server or a particular site encounters a failure or goes offline, users can still access the same data from another server that holds a replica. This redundancy ensures that the database remains available and operational, minimizing downtime and preserving data accessibility even during unexpected disruptions. Such a system is vital for ensuring continuous business operations and service availability.

6.1.2.3 Better Organization

Distributed databases offer structured data management across different organizational units. Each department, branch, or regional office can store and manage its own local data independently, while still being part of the overall enterprise system. This allows for autonomy and efficiency in data handling, while maintaining centralized coordination. It also makes data administration more practical by allowing localized updates, customizations, and access controls without compromising integration with the broader database environment.

6.1.2.4 Flexible Expansion

A distributed database system supports easy and flexible scalability, making it possible to expand the system by adding more servers, storage devices, or nodes as needed. This modular approach to growth allows organizations to scale their infrastructure gradually, in response to increasing data volumes or user demand. Without the need to overhaul

the existing system, new components can be seamlessly integrated, ensuring long-term adaptability and efficient resource utilization.

6.1.2.5 Improved disaster recovery

Distributed databases offer enhanced disaster recovery capabilities by replicating data across multiple nodes and geographic regions. This decentralized architecture ensures high availability and fault tolerance, allowing systems to continue operating even during partial failures. Features like asynchronous replication and integrated backup mechanisms enable efficient and reliable data restoration in the event of disruptions.

6.1.3 Challenges in Distributed Databases

6.1.3.1 Complex Management

Managing a distributed database system involves a high degree of complexity, primarily due to the need to coordinate data across multiple locations. Administrators must handle various technical aspects such as synchronization, network configurations, security protocols, and system integration. Unlike centralized systems, distributed databases require careful planning and advanced management strategies to ensure that all components function cohesively. Monitoring performance, managing backups, handling queries across different servers, and troubleshooting issues across locations demand significant administrative effort and expertise.

6.1.3.2 Data Consistency Issues

Maintaining data consistency across all distributed nodes is one of the major challenges in such systems. When data is replicated or shared across multiple locations, it is essential to ensure that any updates or changes are accurately reflected in all copies. However, due to network delays or system failures, there is a risk of inconsistencies in the stored data, which can lead to errors, duplication, or outdated information. Ensuring synchronous updates and proper conflict resolution mechanisms is critical but requires additional system resources and careful planning.

6.1.3.3 Higher Setup Cost

Implementing a distributed database system typically involves significant initial investment compared to a centralized setup. The infrastructure requires multiple servers, advanced network hardware, data replication tools, and often specialized software to manage distributed transactions. Additionally, the need for skilled personnel to install, configure, and maintain the system contributes to the overall cost. While the long-term benefits may justify the investment, the upfront cost for hardware, connectivity, and system integration can be considerably higher.

6.1.4 Example of a Distributed Database

Consider a large national bank that operates branches in multiple cities across the country. Instead of storing all customer information in a single central database at the bank's headquarters, the bank adopts a distributed database system. In this setup,

each branch maintains a local database containing the information relevant to its own customers, such as account balances, loan records, and transaction history. However, all these local databases are also interconnected through a central coordination system, which ensures data synchronization and seamless communication between branches.

If a customer who opened an account in the Mumbai branch visits the Delhi branch to withdraw money or update account details, the local server in Delhi can access the customer's data instantly from the Mumbai branch or the central system, depending on how the data is distributed or replicated. This setup ensures that the customer experiences real-time access and consistent service, regardless of their location.

Moreover, if the Mumbai branch server goes down due to a technical failure, other branches can still provide services to customers whose data resides there, thanks to replication and redundancy in the distributed system. This ensures high availability, fault tolerance, and better disaster recovery.

This structure enhances performance, as day-to-day operations like deposits or withdrawals are processed locally, reducing network traffic and response time. Meanwhile, centralized coordination ensures data consistency, security policies enforcement, and transaction integrity across the entire banking network.

Thus, the bank benefits from a scalable, reliable, and efficient system, while customers enjoy uninterrupted access to services from any branch location, a perfect example of how distributed databases serve large, geographically spread organizations.

6.1.5 Architectures for parallel Databases

A parallel database system is a type of database that improves performance by processing multiple operations at the same time. It uses multiple processors and storage devices, which allows data to be retrieved, modified, and stored quickly. Parallel databases are commonly used in banking systems, online shopping websites, scientific research, and cloud-based applications where fast data processing is required. The main goal of a parallel database system is to divide the work among different processors so that tasks can be completed faster. This process is similar to a school project where different students are assigned different parts of the work, making it easier and quicker to complete.

Parallel database systems are designed to improve throughput, speed, and scalability by executing multiple operations simultaneously. The underlying architecture of a parallel database system plays a crucial role in determining how processing units (nodes) interact with memory and storage. The three primary parallel database architectures are: Shared Memory, Shared Disk, and Shared Nothing. These three types of parallel database architectures explained below.

6.1.5.1 Shared Memory Architecture

In Shared Memory Architecture, all processors in the parallel database system have access to a common shared memory and a single shared disk. This setup allows all processors to work on different tasks while reading and writing to the same memory space, making communication between processors fast and simple. Since the data

resides in a single location, managing it is straightforward. However, as the number of processors increases, contention for shared resources (memory and disk) can reduce performance, limiting the system's scalability. This architecture is most suitable for symmetric multiprocessing (SMP) systems and smaller-scale parallel databases where ease of implementation is a priority. Fig. 6.1.1 shows the block diagram of shared-memory system.

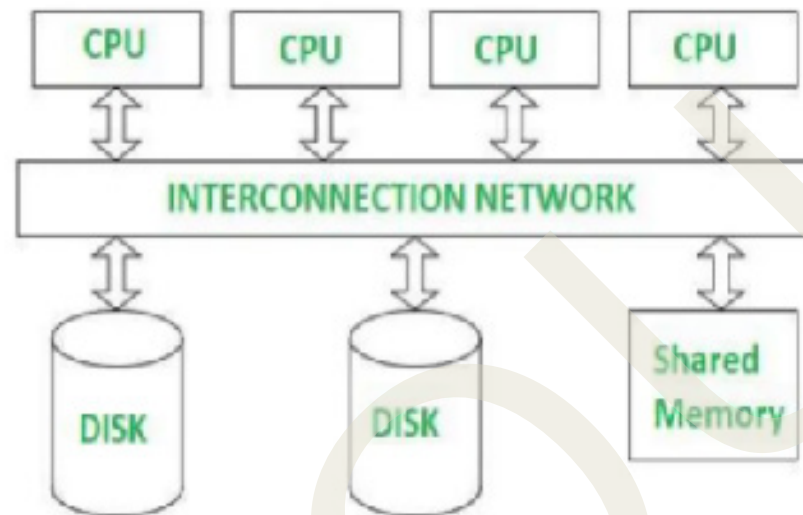


Fig. 6.1.1 Shared Memory System

Characteristics:

- ◆ All processors share the same memory and I/O subsystem.
- ◆ Suitable for Symmetric Multiprocessing (SMP) systems.
- ◆ Ideal for smaller-scale parallel transaction processing environments.

Benefits:

- ◆ Simple to manage and implement.
- ◆ Data sharing is easy and efficient, making inter-process communication fast.
- ◆ Useful for workloads where data partitioning isn't necessary.

Limitations:

- ◆ Scalability is limited : As more processors are added, contention for shared resources increases.
- ◆ Risk of bottlenecks in memory access and I/O channels.
- ◆ Failure in shared memory can affect the entire system.

6.1.5.2 Shared Disk Architecture

In Shared Disk Architecture, each processor or node has its own private memory but shares access to the same disk storage with other nodes. This means that although processing is distributed across nodes, the data remains centrally stored. Such architecture allows better fault tolerance, if one processor fails, others can still access the shared data and continue operations. However, shared access to the disk requires complex coordination mechanisms such as distributed locking and concurrency control to ensure consistency. While this architecture offers moderate scalability, it can still face performance bottlenecks due to shared disk I/O access. Shared disk systems are commonly used in high-availability clustered databases like Oracle RAC. Fig. 6.1.2 shows the block diagram of shared-disk system.

Characteristics:

- ◆ Local memory per processor, but shared access to disk storage.
- ◆ Data is centrally stored, but processing is distributed.
- ◆ Requires concurrency and locking control mechanisms to ensure consistency.

Benefits:

- ◆ High availability : If one node fails, others can continue processing using the same shared data.
- ◆ Easier to scale compared to shared memory systems.
- ◆ Centralized data makes data management and consistency easier.

Limitations:

- ◆ Requires complex synchronization and lock management across nodes.
- ◆ Disk I/O bottlenecks may arise under heavy load.
- ◆ More sophisticated software is required to handle coordination.

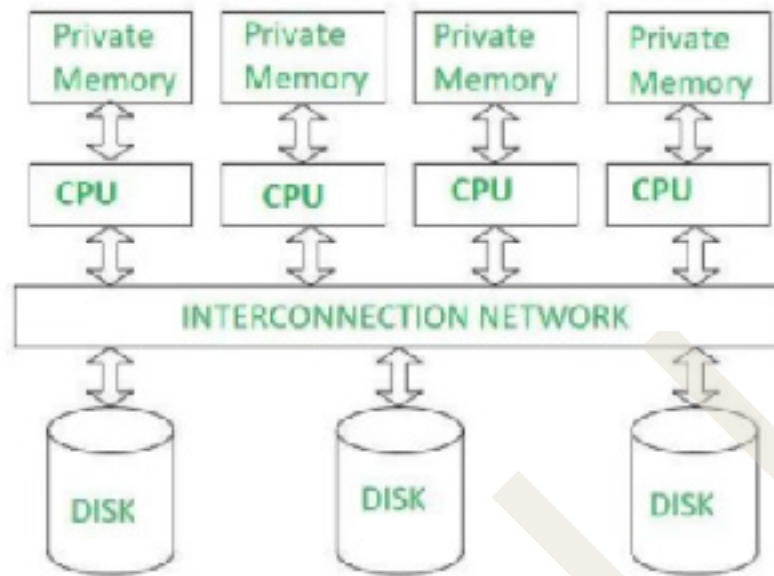


Fig. 6.1.2 Shared-Disk System

6.1.5.3 Shared Nothing Architecture

Shared Nothing Architecture is the most scalable and fault-tolerant among the three. Here, each node has its own independent memory and disk storage, and there is no sharing of data or resources among nodes. Data is partitioned across nodes, and each node handles processing tasks related to its own portion of the data. Nodes communicate with each other only when necessary, typically through a network. This decentralized model eliminates resource contention and allows seamless horizontal scaling, new nodes can be added without disrupting the system. Although this architecture offers excellent performance and scalability, it requires careful data distribution and sophisticated query coordination. Shared Nothing Architecture is widely used in large-scale data warehousing and big data platforms such as Hadoop, Cassandra, and Amazon Redshift. Fig. 6.1.3 shows the block diagram of shared nothing architecture.

Characteristics:

- ◆ Fully decentralized architecture.
- ◆ Data is partitioned across nodes : Each node handles a subset of data.
- ◆ Most scalable architecture among the three.

Benefits:

- ◆ Excellent horizontal scalability : More nodes can be added without performance degradation.
- ◆ Fault isolation : Failure of one node doesn't impact the others.
- ◆ Highly suited for large-scale data warehousing and distributed analytics.

Limitations:

- ◆ Requires efficient partitioning and query routing mechanisms.
- ◆ Coordination and communication overhead can arise in complex queries.
- ◆ Maintaining global data consistency can be challenging.

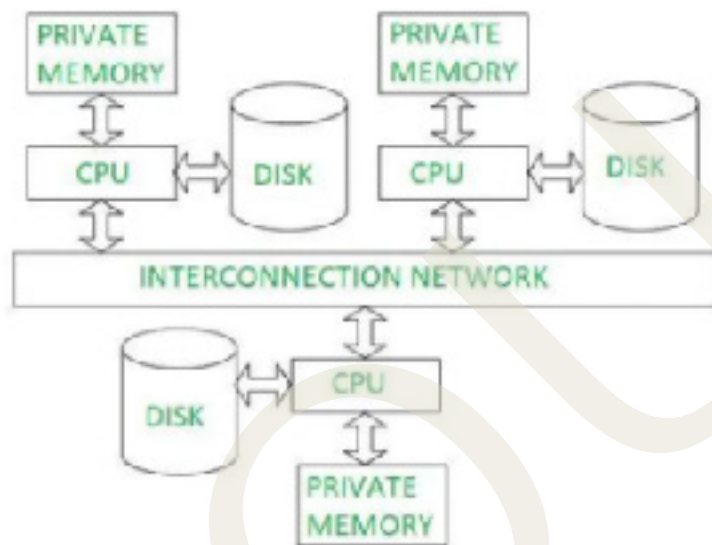


Fig. 6.1.3 Shared nothing architecture

6.1.6 Comparison of Parallel Database Architectures

The following table compares the three main architectures used in parallel database systems: Shared Memory, Shared Disk, and Shared Nothing. Each architecture has distinct characteristics, benefits, and limitations, especially in terms of scalability, fault tolerance, data sharing, and complexity.

Table 6.1.1 Parallel Database Architecture systems- comparison

Feature	Shared Memory Architecture	Shared Disk Architecture	Shared Nothing Architecture
Memory and Storage Access	All processors share the same memory and disk	Processors have private memory but share disk	Each processor has its own memory and disk
Data Sharing	Easy and fast inter-process communication	Shared disk allows data access by all nodes	No data sharing; nodes manage their own data
Scalability	Limited due to contention for shared resources	Moderate; disk access may become a bottleneck	High; new nodes can be added easily

Fault Tolerance	Low; memory failure can affect entire system	Moderate; nodes can continue if one fails	High; node failures do not impact others
Complexity	Low; simple to implement and manage	Moderate; requires coordination mechanisms	High; needs sophisticated partitioning and routing
Use Cases	Small-scale SMP systems	High-availability clusters (e.g., Oracle RAC)	Large-scale data warehouses (e.g., Hadoop, Redshift)

How Parallel Database Systems Work?

In a parallel database system, multiple processors work together to handle queries and transactions. Each processor may have its own memory and storage or may share resources with others. When a user requests data, the system divides the task into smaller parts and processes them at the same time, increasing efficiency. Parallel databases help improve speed, reliability, and scalability, meaning they can handle growing amounts of data without slowing down. These systems are useful for applications that require quick decision-making and real-time processing.

Parallel databases provide several advantages over traditional database systems. Since multiple processors work together, tasks are completed faster. The system can grow by adding more processors and storage, making it highly scalable. Additionally, even if one processor fails, others can continue working, ensuring high availability. Efficient load balancing is another benefit, as work is distributed among multiple processors, preventing overload on a single system. These advantages make parallel databases essential for modern applications, including financial transactions, healthcare systems, social media platforms, and scientific research, where handling massive amounts of data efficiently is important.

Parallel database systems play a crucial role in improving the performance, reliability, and scalability of database applications. As technology advances, parallel databases will continue to evolve, supporting even faster and more reliable data management solutions.

6.1.7 Parallel Query evaluation

Imagine you are solving a big math problem with your friends. Instead of one person doing all the work, you split the problem into smaller parts, and each friend works on a different part at the same time. This helps you solve the problem faster. This is similar to how databases use Parallel Query Evaluation.

In large databases, users run many queries to get information. If a single computer processes all queries one by one, it takes a long time. Instead, databases use multiple computers working together, just like a group of friends solving a big problem. This is called Parallel Query Processing.

6.1.7.1 How Query Processing Works in Distributed Databases?

When we use a distributed database system, the data is not stored in just one place—it is spread across different computers or sites connected by a network. To get the correct answer when a user sends a query (a request to get data), the system needs to collect and process information from different sites. This process is called query processing in a distributed database management system (DBMS).

Unlike normal databases that work from a single computer (centralized DBMS), a distributed DBMS has to transfer data between different sites, which can increase the communication cost (the effort and time it takes to send data across the network). If the computers are connected with a high-speed network, this cost is low. But if the network is slower, the communication cost becomes a big concern.

The main steps in query processing include:

1. Breaking down the query into smaller tasks.
2. Sending tasks to different sites where the data is stored.
3. Processing tasks in parallel (at the same time).
4. Collecting and combining results to show the final answer to the user.

Cost of Data Transfer in Distributed Queries

When a query is executed in a distributed system, data sometimes needs to be moved from one site to another before the query can be processed completely. This movement of data involves cost, called data transfer cost. For example, if a user at Site 3 asks a question, but the required data is stored at Site 1 and Site 2, the data needs to be sent over the network.

There are three main strategies to process this:

1. Send all data from Site 1 and Site 2 to Site 3 and process it there.
2. Send data from Site 1 to Site 2, process it there, then send the result to Site 3.
3. Send data from Site 2 to Site 1, process the query there, and then send the result to Site 3.

The choice depends on how large the tables are and how fast data can be sent between sites. The formula for calculating the cost is:

Data transfer cost = $C \times \text{Size of data in bytes}$ (C is the cost per byte of data sent.)

For example let's say we have two tables:

- ◆ EMPLOYEE table at Site 1 (each record is 60 bytes, and there are 1000 records).
- ◆ DEPARTMENT table at Site 2 (each record is 30 bytes, and there are 50 records).

A query is asked at Site 3 to find the names of employees and their department names. Since neither table is at Site 3, data needs to be moved to site 3.

Three ways to do this:

1. Move both tables to Site 3 and process there. Total data moved: 61,500 bytes.
2. Move EMPLOYEE to Site 2, join the tables, then send result to Site 3: 120,000 bytes.
3. Move DEPARTMENT to Site 1, process there, send result to Site 3: 61,500 bytes.

So, strategy 1 or 3 is better if we want to minimize data transfer.

Using Semi-Join to Reduce Data Transfer

A semi-join is a smart way to reduce the amount of data that needs to be sent over the network. Instead of sending the full table, we send only the parts needed to match data from the other table.

For example, we can:

- ◆ Select only NAME and DID from the EMPLOYEE table (instead of the full table), and send it to Site 3. Size = 30,000 bytes.
- ◆ Send the full DEPARTMENT table to Site 3. Size = 1,500 bytes.
- ◆ Then, join both at Site 3.

This strategy brings down the total data transfer cost to only 31,500 bytes, which is much better than the other methods.

In distributed databases, query processing requires smart planning to reduce time and cost. By choosing the right method—like using semi-joins—we can process queries faster and avoid sending large amounts of unnecessary data across the network.

6.1.7.2 Types of Parallel Query Processing

Parallel query processing happens in two main ways:

Inter-Query Parallelism (Parallel Execution of Multiple Queries):

Inter-query parallelism refers to a scenario where multiple queries are executed simultaneously, with each query being processed independently by a different processor or node in a parallel database system. This approach enhances the overall system throughput by handling more user requests in parallel, especially in multi-user environments.

Imagine a database server receiving several queries at the same time from different users. Instead of processing them one after another (sequentially), the system assigns each query to a different processor or computing unit. These processors work independently on their assigned queries without interfering with one another, resulting in faster response times and better resource utilization.

This is similar to a classroom where each student is given a different question to solve. Since every student works on a separate problem, all questions can be solved at the same time, speeding up the overall completion process. In this analogy, students represent processors, and the questions represent individual queries.

Benefits:

- ◆ Increases system throughput.
- ◆ Suitable for multi-user environments where many queries are submitted concurrently.
- ◆ Improves overall system efficiency.

Intra-Query Parallelism (Parallel Execution Within a Single Query):

Intra-query parallelism is a technique where a single complex query is broken down into smaller tasks or operations, and these tasks are distributed across multiple processors to be executed in parallel. The goal is to reduce the execution time of that individual query by performing multiple operations simultaneously.

When a query involves a large amount of data, such as scanning a massive table, joining multiple datasets, or performing aggregations, it is inefficient to process it on a single processor. Instead, the system divides the query into sub-tasks (such as scanning different partitions of a table or computing aggregations on subsets of data), assigns each sub-task to a different processor, and then combines the results to produce the final output.

Think of a group of students working together to solve a single, lengthy math problem. One student calculates the first step, another works on the second step, while a third student completes the final step. By dividing the work, the entire solution is achieved much faster than if a single student had solved it alone. Here, the single query is split into steps, and each step is handled concurrently by a separate processor.

Types of Intra-Query Parallelism:

1. **Intra-Operation Parallelism** : Parallel execution of operations like sorting, scanning, or joining within a single query.
2. **Inter-Operation Parallelism** : Executing different operations (e.g., selection, join, aggregation) in a query plan concurrently when they are not dependent on each other.
3. **Pipeline Parallelism** : The output of one operation is passed immediately to the next operation without waiting for the entire operation to complete, creating a pipeline of tasks.

Benefits:

- ◆ Significantly reduces query response time.
- ◆ Best suited for large, complex queries in data warehousing and analytical processing.



- ◆ Makes efficient use of parallel hardware resources.

Parallel query processing is a method used in database systems to execute queries more efficiently by utilizing multiple processors or computers simultaneously. The process begins with breaking down the query into smaller, independent tasks. This step is crucial because large queries often involve complex operations such as table scans, joins, filtering, sorting, or aggregations. The database system analyzes the query and divides it into sub-tasks that can be processed separately without affecting the correctness of the final result. Once the query is decomposed, these tasks are assigned to different computers or processing nodes within the parallel database architecture. Each node is responsible for handling a specific portion of the overall query.

Following task allocation, the system initiates parallel processing, where each computer works on its assigned task simultaneously and independently. This concurrent execution significantly reduces the overall time required to complete the query, making it much faster than traditional sequential processing. After all nodes complete their respective parts, the system moves to the final stage, which is combining the results. In this step, the partial results generated by each node are gathered, merged, and organized into a cohesive output that represents the final answer to the original query. This process ensures not only speed and efficiency but also maintains the accuracy and completeness of query results. Parallel query processing is especially beneficial in environments where databases are large and queries are data-intensive, such as in data warehousing, business intelligence, and big data analytics systems.

6.1.7.3 Advantages of Parallel Query Processing

- ◆ **Faster Performance:** Since multiple computers work together, queries are processed quickly.
- ◆ **Better Use of Resources:** Instead of one computer doing all the work, the load is shared.
- ◆ **Scalability:** More computers can be added to process more data efficiently.

6.1.7.4 Challenges in Parallel Query Processing

- ◆ **Data Distribution:** The system must decide how to divide the data among computers for faster processing.
- ◆ **Synchronization:** All computers must work together correctly, and their results must be combined properly.
- ◆ **Communication Overhead:** Computers must communicate with each other, which can slow down performance if not managed well.
- ◆ **Issues related to load balancing:** In large-scale parallel query processing, load balancing ensures tasks are evenly distributed across nodes. Challenges like data skew, varying query complexity, and resource differences can lead to some nodes being overloaded while others are underutilized. This causes delays, poor performance, and inefficient resource use. Dynamic task

allocation and smart partitioning help mitigate these issues. Effective load balancing is essential for maximizing system efficiency.

Parallel Query Processing is a powerful technique used in databases to process queries faster. By splitting work among multiple computers, databases can handle large amounts of data efficiently. This is especially useful for big companies, websites, and applications that process a lot of data every second.

6.1.8 Parallelizing Individual Operations

When a computer performs tasks, it follows several steps. Some tasks can be done at the same time, while others must be done one after another. Parallelizing individual operations means breaking down a big task into multiple smaller steps and running those steps simultaneously.

In databases and software programs, a single operation can sometimes take a lot of time. Instead of waiting for one task to finish before starting the next, the computer splits the task into smaller steps and processes them in parallel to complete the job faster.

6.1.8.1 Example of Parallel Execution

Imagine a bakery that has to bake 100 cupcakes. If only one baker does all the work, mixing ingredients, pouring batter, baking, and decorating—it will take a long time. But if different workers handle different steps at the same time, such as one mixing the batter, another pouring it, and another baking, they will finish much faster.

Similarly, in a computer, parallelizing tasks means different processors work on different parts of a problem at the same time, making the system more efficient.

6.1.8.2 Why is Parallelizing Important?

Parallelizing operations helps in many ways:

1. **Increases Speed** : Tasks are completed much faster than when processed one at a time.
2. **Improves Efficiency** : The computer can use its power more effectively by handling multiple tasks at once.
3. **Handles Large Data** : When there is a lot of information to process, parallel computing helps divide and conquer the problem.
4. **Reduces Waiting Time** : When tasks run in parallel, the user does not have to wait too long for results.

To understand how parallel computing works, let's look at the step-by-step process:

1. **Task Division** : A big task is divided into smaller subtasks.
2. **Parallel Execution** : These smaller tasks are assigned to different processors to work on them at the same time.



3. **Synchronization** : After processing, the results from different tasks are combined to get the final answer.

For example, when you watch a video on YouTube, the system does multiple tasks in parallel, loading the video, buffering data, and adjusting quality based on internet speed, all happening at the same time.

6.1.8.3 Uses of Parallel Computing

Parallel computing is used in many real-world applications:

1. **Online Search Engines (Google, Bing, Yahoo)** : These platforms search through millions of web pages simultaneously to give you quick results.
2. **Weather Forecasting** : Computers process weather data from thousands of locations at the same time to predict future climate conditions.
3. **Gaming and Graphics Processing** : Modern video games use parallel computing to render realistic images and animations quickly.
4. **Space Exploration** : NASA and other space agencies use parallel computing to analyze large sets of data from space telescopes.
5. **Healthcare and Medicine** : Medical researchers use parallel computing to analyze genetic data and develop new medicines.

6.1.8.4 Challenges of Parallel Computing

Although parallel computing is powerful, it also has some challenges:

1. **Complexity** : Dividing tasks and managing multiple processors requires advanced programming.
2. **Synchronization Issues** : Sometimes, different parts of the task finish at different times, which can cause delays.
3. **Hardware Requirements** : Parallel computing requires multiple processors or high-performance computing systems.

Despite these challenges, parallel computing is becoming more advanced and is used in many modern technologies.

Parallelizing individual operations is an important technique that allows computers to process tasks more efficiently by splitting them into smaller steps and executing them at the same time. It is used in search engines, video games, weather forecasting, healthcare, and many other fields. Understanding parallel computing helps us appreciate how modern computers perform complex tasks quickly and efficiently.

By learning about parallel computing, students can develop an interest in computer science and explore careers in technology, software development, and artificial intelligence.

6.1.9 Overview of Distributed Database

A Distributed Database System (DDBS) is a type of database that stores data across multiple computers instead of just one. This means that the database is spread over different locations, but it still appears as a single system to users.

Imagine a big library with books in many different branches of a city. If a reader wants a book, they can get it from the nearest branch instead of traveling to the main library. Similarly, a distributed database allows users to access data from the nearest server instead of going to a central computer.

Distributed databases are used because they offer many benefits:

1. **Faster Access** : Since data is stored in different locations, users can access it quickly from the nearest server.
2. **More Reliability** : If one server fails, others can still work, so data is not lost.
3. **Scalability** : More computers can be added to handle more data as needed.
4. **Better Performance** : Workload is shared among multiple computers, reducing delays.

Challenges of Distributed Databases

While they have many advantages, distributed databases also come with challenges:

- ♦ **Data Synchronization** : Keeping all copies of data updated can be difficult.
- ♦ **Security Issues** : Data stored in different places may be harder to protect.
- ♦ **Complexity** : Managing multiple servers and ensuring they work together smoothly is more challenging than using a single database.

Examples of Distributed Databases

- ♦ **Banking Systems** : Banks use distributed databases to store customer information across different branches.
- ♦ **Online Shopping** : E-commerce websites store customer orders and product information in multiple locations to ensure fast service.
- ♦ **Social Media Platforms** : Sites like Facebook and Instagram use distributed databases to store user profiles, messages, and media files.

Distributed databases help organizations manage data more efficiently by spreading it across multiple locations. They provide faster access, reliability, and better performance but require careful management to handle challenges like security and synchronization.

Recap

- ◆ A distributed database stores data on multiple computers instead of just one.
- ◆ These computers are connected through a network and share data with each other.
- ◆ Data distribution means storing parts of data in different places to improve speed and balance the load.
- ◆ Replication copies data to multiple locations so it is still available if one location fails.
- ◆ Even though the data is stored in many places, users can access it as if it's from one system.
- ◆ Distributed databases are reliable because if one computer fails, others can still work.
- ◆ These systems are scalable, meaning new computers can be added easily to handle more data.
- ◆ They offer faster access because users get data from the closest server.
- ◆ Distributed systems allow different departments or branches to manage their own data independently.
- ◆ Parallel databases use many processors to work on data at the same time for faster performance.
- ◆ There are three types of parallel database systems: Shared Memory, Shared Disk, and Shared Nothing.
- ◆ In Shared Memory, all processors use the same memory and disk, but scalability is limited.
- ◆ In Shared Disk, each processor has its own memory but shares the disk, improving availability.
- ◆ Shared Nothing architecture is most scalable, with each processor having its own memory and disk.
- ◆ Parallel query processing means breaking one big query into smaller parts and running them together.
- ◆ Inter-query parallelism runs many queries at the same time on different processors.
- ◆ Intra-query parallelism splits one query into parts, with each part handled by a different processor to save time.

Objective Type Questions

1. Which type of the database is a distributed database?
2. What is the main goal of a parallel database system?
3. In a distributed database system, the need to move data between sites is called.
4. Which architecture has each node with its own memory and storage without sharing anything?
5. Is better organization and improved reliability are benefits of distributed databases?
6. In Shared Disk Architecture, nodes share:
7. Data transfer cost in distributed query processing is calculated as:
8. Which of the following is a technique to reduce data transfer in distributed queries?
9. Which type of query parallelism runs multiple queries at the same time on different processors?
10. True or False: Shared memory architecture is most suitable for large-scale distributed systems.
11. Which architecture provides the best scalability in a parallel database system?
12. Which challenge is associated with ensuring all copies of data in a distributed database are updated correctly?
13. What is the main advantage of using parallel query processing?
14. Is cross-operation a type of intra-query parallelism?
15. Name some real-world application of parallel computing.
16. True or False: In Shared Nothing architecture, nodes do not communicate with each other at all.
17. What does scalability in distributed databases allow?
18. Which operation involves combining partial results from multiple processors in parallel query processing?

Answers to Objective Type Questions

1. Distributed in nature
2. Performance
3. Communication
4. Shared Nothing
5. Yes
6. Disk
7. $C \times \text{Size of data in bytes}$
8. Semi-Join
9. Inter-query
10. False
11. Shared Nothing
12. Consistency
13. Speed
14. No
15. Weather forecasting, online search, gaming
16. False
17. Expansion
18. Merging and organizing

Assignments

1. Explain the concept of a Distributed Database System with suitable examples.
2. Discuss the key features of Distributed Databases such as data distribution, replication, independence, reliability, and scalability.
3. What are the major advantages and challenges of Distributed Database Systems?
4. Describe the working of a Distributed Database in a real-world scenario such as a banking system.
5. Explain the different architectures of Parallel Database Systems: Shared Memory, Shared Disk, and Shared Nothing.

6. Describe the process of Parallel Query Processing in a database system.
7. Differentiate between Inter-Query and Intra-Query Parallelism with appropriate examples.
8. What is a Semi-Join in Distributed Query Processing? How does it help in reducing data transfer costs?

Suggested Reading

1. Ceri, S., & Pelagatti, G. (1984). *Distributed databases: Principles and systems*. McGraw-Hill.
2. DeWitt, D., & Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 85–98. <https://doi.org/10.1145/129888.129892> (If DOI not available, omit link)
3. Elmasri, R. (2008). *Fundamentals of database systems* (6th ed.). Pearson Education India.
4. Lentz, A., Zaitsev, P., Tkachenko, V., Zawodny, J., Balling, D., & Schwartz, B. (2008). *High performance MySQL: Optimization, backups, replication, and more* (2nd ed.). O'Reilly Media.
5. Özsu, M. T., & Valduriez, P. (1999). *Principles of distributed database systems* (Vol. 2). Prentice Hall.
6. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). *Database system concepts* (6th ed.). McGraw-Hill.
7. Van Steen, M. (2002). *Distributed systems: Principles and paradigms*. Pearson Education.

Reference

1. DeWitt, D., & Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 85–98. <https://doi.org/10.1145/129888.129892> (Include DOI if available, otherwise omit the link)
2. Lentz, A., Zaitsev, P., Tkachenko, V., Zawodny, J., Balling, D., & Schwartz, B. (2008). *High performance MySQL: Optimization, backups, replication, and more* (2nd ed.). O'Reilly Media.
3. Van Steen, M. (2002). *Distributed systems: Principles and paradigms*. Pearson Education.

Unit 2

Data Fragmentation and Replication

Learning Outcomes

Upon the completion of this unit, the learners will be able to:

- ♦ to understand the concept of data fragmentation and its importance in optimizing distributed database performance
- ♦ to identify the role of data replication in ensuring data availability, consistency, and fault tolerance across distributed systems
- ♦ to analyse various data allocation techniques and their impact on system efficiency, cost, and query performance
- ♦ to apply distributed database design concepts to improve efficiency and scalability

Prerequisites

To understand Data Fragmentation and Replication in Distributed Database Design, one must grasp key database and distributed system concepts through real-world scenarios. For instance, knowledge of Database Management Systems (DBMS) is crucial, as seen in Amazon's e-commerce platform, where SQL and ACID properties ensure order consistency. Understanding distributed databases helps in designing global systems like Netflix, which serves content from multiple locations for faster access. Data storage and access methods play a role in services like Google Maps, where location data is partitioned to allow quick retrieval. Familiarity with networking and distributed systems is necessary, as seen in WhatsApp's message replication across multiple servers for real-time delivery. Concepts of data consistency and concurrency Control are vital in banking applications like PayPal, where transaction integrity is maintained to prevent double withdrawals. Finally, understanding performance optimization techniques, such as caching and load balancing, helps optimize platforms like YouTube, ensuring smooth video streaming without buffering. These real-world applications highlight the importance of fragmentation, replication, and allocation techniques in distributed databases.

Keywords

Horizontal Fragmentation, Vertical Fragmentation, Data Allocation, Distributed Database Design



Discussion

Distributed Database System (DDBS) is a collection of multiple interconnected databases distributed across various locations. The primary goals of distributed databases are improved performance, reliability, availability, and scalability. To achieve efficient data distribution, two techniques are employed: data fragmentation and data replication. Data fragmentation involves breaking down the database into smaller, more manageable pieces, allowing for better organization and access. Data replication, on the other hand, maintains multiple copies of data to enhance reliability and availability. By combining these techniques, data can be distributed more efficiently, ensuring improved performance and accessibility. For instance, e-commerce websites like Amazon, Flipkart, and eBay deal with large amounts of data, including customer information, order history, and product details. To ensure efficient data management and scalability, these websites employ data fragmentation and replication techniques.

6.2.1 Data fragmentation

"Imagine you are trying to find a specific toy in a big box. It would take a long time to search through the entire box! But if you organize the toys into smaller boxes, you could find the toy much faster.". This is similar to how data fragmentation works in databases, where large amounts of data are broken down into smaller, more manageable pieces. By fragmenting data, databases can retrieve information more quickly and efficiently.

Data fragmentation enables faster data retrieval and management. Furthermore, fragmenting data makes it easier to share with others and store in various locations. It involves breaking down a large database into smaller, more manageable fragments.

A database of customers with attributes name, address, phone number, cust_id, order_date and order_amount can be fragmented into two relations:

- Customer_info (name, address, phone number)
- Order_Detail (cust_id, order_date, order_amount)

6.2.1.1 Data Fragmentation Techniques

Fragmentation can be of following four types:

- 1. Horizontal Fragmentation:** In a horizontal fragmentation, a relation is divided into smaller subsets of tuples, with each subset stored on a separate computer. The following table Employee demonstrates the concept of horizontal fragmentation.

Attributes (Name, Department, Salary)

Input:

Name	Department	Salary
John	Sales	50000
Jane	Marketing	60000
Bob	Sales	40000
Alice	Marketing	70000

Horizontally fragmented relations:

SELECT * FROM Employee WHERE Salary < 55000;

Output:

Name	Department	Salary
John	Sales	50000
Bob	Finance	40000

A Customer table can be fragmented by region:

Fragment 1: SELECT * FROM Customer WHERE Region = 'North';

Fragment 2: SELECT * FROM Customer WHERE Region = 'South';

2. **Vertical Fragmentation:** Vertical fragmentation involves dividing a relation into smaller relations based on columns. The following table Employee demonstrates the concept of horizontal fragmentation.

Attributes (Name, Address, Phone, Salary)

Input:

Name	Address	Phone	Salary
John	123 Main St	123-456-7890	50000
Jane	456 Elm St	987-654-3210	40000

Vertically fragmented relations:

Select Name, Address from Employee;

Output:

Name	Address
John	123 Main St
Jane	456 Elm St

A Customer table can be fragmented into:

- ◆ Fragment 1: CustomerID, Name, Address
- ◆ Fragment 2: CustomerID, Phone, Email

3. **Mixed or Hybrid Fragmentation:** Hybrid fragmentation, which combines horizontal and vertical fragmentation, offers greater flexibility in data distribution. This approach allows for a more granular and efficient placement of data across different nodes, optimizing performance and resource utilization in distributed database systems.

Input:

Name	Department	Salary	Job History
John	Sales	50000	Sales, Marketing
Jane	Marketing	60000	Marketing, Sales
Bob	Sales	40000	Sales
Alice	Marketing	70000	Marketing

Mixed fragmented relations:

Name	Salary	Job History
John	50000	Sales, Marketing
Bob	40000	Sales

A Customer table can be fragmented: Horizontally by region and vertically by attributes:

Fragment 1: SELECT CustomerID, Name, Address FROM Customer WHERE Region = 'North';

Fragment 2: SELECT CustomerID, Phone, Email FROM Customer WHERE Region = 'South';

4. Derived Fragmentation:

1. A special form of horizontal fragmentation.
2. Fragments of one relation are derived from the fragments of another relation.
3. Common in parent-child relationships.
4. Rows are divided based on conditions applied to another related table (e.g., foreign key relationships).

Example: If Orders is derived from Customers, the Orders fragments can be derived based on the Customers fragments.

6.2.1.2 Advantages of Data Fragmentation

- ◆ **Improved performance:** Reduces the amount of data accessed by queries, leading to faster query execution.
- ◆ **Enhanced availability:** If one fragment is unavailable, other fragments can still be accessed.
- ◆ **Better manageability:** Smaller fragments are easier to maintain, back up, and recover.
- ◆ **Localized data access:** Data can be stored closer to the users who need it, reducing network latency.

6.2.1.3 Disadvantages of Data Fragmentation

- ♦ **Complexity:** Managing fragments and ensuring consistency across them can be challenging.
- ♦ **Reconstruction overhead:** Combining fragments to reconstruct the original table may require additional processing.
- ♦ **Data integrity:** Ensuring referential integrity and consistency across fragments can be difficult.

6.2.2 Data Replication

Data replication in DBMS refers to the process of creating and maintaining multiple copies of the same data across different locations or nodes in a distributed system. Replication is used to improve data availability, fault tolerance, and performance. It is a key technique in distributed databases and systems where data needs to be accessed by users or applications from multiple locations.

6.2.2.1 Key Features of Data Replication

1. **Higher Data Availability:** Data replication helps ensure that information is always accessible by maintaining multiple copies of the same data across different locations. Even if one server or system fails due to a hardware issue or network problem, other copies remain available, reducing the risk of data unavailability.
2. **Better System Performance:** Since the same data is available at various sites, users can access it from the nearest location, which reduces network delays and enhances query response time. This makes data retrieval faster and more efficient.
3. **Enhanced System Scalability:** Replication supports scalable database systems by distributing data across different nodes. This allows the system to handle more users and larger volumes of data efficiently, improving overall performance as the workload increases.
4. **Greater Fault Tolerance:** By keeping redundant copies of data, replication ensures that data remains available even if one part of the system fails. This makes the system more resilient to errors or failures and helps maintain smooth operations.
5. **Improved Data Locality:** Replication places data closer to the users or applications that need it. This reduces network traffic and improves access speed, especially in systems with users spread across different geographic locations.

6.2.2.2 Types of Data Replication

1. **Transactional Replication:** In this method, each user first receives a complete copy of the database, and afterward, all changes or updates are sent to them as they happen, in the same sequence as the source. This ensures

transactional consistency, meaning the order and accuracy of data changes are preserved. It is commonly used in server-to-server environments where high reliability and real-time updates are important. Unlike basic data copying, this method replicates every change consistently and accurately.

2. **Snapshot Replication:** Snapshot replication captures the entire database at a specific point in time and sends that version to other users. It does not track or apply continuous updates. This method is best suited for scenarios where data changes are rare or infrequent. Though it is slower compared to transactional replication, it is useful for initial database synchronization between the source (publisher) and other systems (subscribers).
3. **Merge Replication:** In merge replication, data from multiple sources is combined into a single database. This method is more complex because it allows both the publisher and the subscriber to make independent changes to their local copies. These changes are then merged into a unified database. Merge replication is typically used in server-to-client environments where two-way communication and updates are needed. It is ideal for mobile or offline users who work with local data and later synchronize it with the central system.

6.2.2.3 Replication Strategies

1. Full Replication

The entire database is replicated across all nodes. Full replication involves storing a complete copy of the entire database at every site within the distributed system. This approach significantly enhances data availability, as the system can continue functioning even if several sites fail—only one active site is needed to access the data. It also allows for faster query execution, since queries can be handled locally at any site, minimizing the need for data transfers. However, this scheme comes with challenges. Maintaining consistency across all replicas is complex, as every data update must be propagated to all sites. This leads to slower update operations and can create difficulties in achieving concurrency control. Additionally, more storage space is required to store identical data copies at multiple locations, and updating all replicas makes the system more expensive to maintain. Despite these limitations, full replication improves data reliability, supports multiple users efficiently, and minimizes data movement by keeping data close to where transactions are executed.

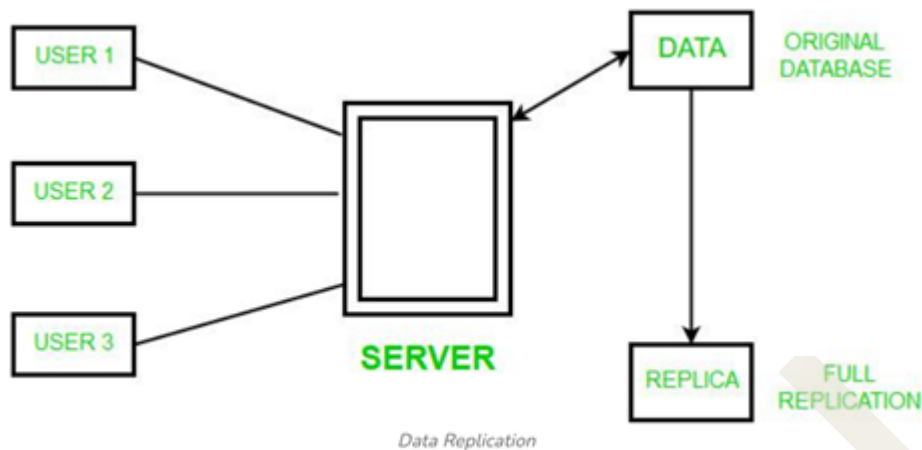


Fig. 6.2.1 Full Replication

Advantages

- ◆ High availability and fault tolerance.
- ◆ Improved read performance as data is locally available.

Disadvantages

- ◆ High storage and maintenance overhead.
- ◆ Slower write performance due to updating all replicas.

2. Partial Replication

Only a subset of the database is replicated across nodes. Partial replication offers a balanced approach, where only some parts (fragments) of the database are replicated across different sites based on their importance and frequency of access. This helps reduce storage costs while still improving data availability and performance where needed. In this scheme, important or frequently accessed fragments may have multiple copies, while less critical data may exist only at a single site. Partial replication combines the advantages of full and no replication, providing both efficiency and flexibility. However, it still requires careful design and planning to ensure that data consistency and synchronization are maintained effectively across the system.

Advantages

- ◆ Reduced storage and maintenance overhead.
- ◆ Improved performance for frequently accessed data.

Disadvantages

- ◆ Risk of data unavailability if a node with a specific fragment fails.

3. No Replication

No copies of the data are maintained; each node stores unique data. In a no replication scheme, each data fragment is stored at only one site, meaning there are no duplicate

copies of data. The main advantage of this method is that concurrency management becomes simpler, as updates are made at only one location. It also requires less storage space and makes data recovery straightforward, since the data resides in a single place. However, this scheme suffers from low data availability, because if the site storing a particular fragment goes down, the data becomes inaccessible. Also, query performance can degrade when multiple users try to access the same site simultaneously, leading to system bottlenecks.

Advantages

- ◆ Minimal storage and maintenance overhead.

Disadvantages

- ◆ Low fault tolerance and availability.

6.2.2.4 Other Types of Data Replication

Data replication is the process of copying and maintaining database objects, like tables, in multiple databases that may reside on the same or different servers. The replication model determines how data is shared and synchronized between systems. Here are the some of other types of replication methods:

1. **Master-Slave Replication:** In this model, one server is assigned the role of master, which handles all data write and update operations. The other servers, known as slaves, receive replicated copies of the data from the master but do not accept direct write operations. Slaves are used primarily for reading data.
2. **Multi-Master Replication:** Here, multiple servers can act as masters, allowing write operations to occur on any server. Changes made to one server are automatically synchronized with all other servers, ensuring consistency across the system. This method supports collaborative data entry across multiple locations.
3. **Peer-to-Peer Replication:** In peer-to-peer replication, every server has equal roles—each can act as both a source and a receiver of data. Data is shared among all servers equally, and updates are replicated across all nodes in a decentralized, bidirectional manner.

Table 6.2.1 Comparison of other replication architectures

Feature / Architecture	Master-Slave (Single-Leader)	Multi-Master (Multiple-Leaders)	Peer-to-Peer (Masterless)
Write Operations	All writings directed to the Master. Slaves are read-only.	Any Master can accept writes.	Any peer can accept writes.
Data Flow	Unidirectional	Bidirectional	Bidirectional

Read Operations	Master and Slaves can serve reads.	All Masters can serve reads.	All peers can serve reads.
Setup Complexity	Relatively Simple	Moderate to High	High
Conflict Handling	Not required	Required due to possible data conflicts	Required for simultaneous updates
Example	News website users read from slaves	Banking system with multiple branches	Google Docs or blockchain-style sync

6.2.2.5 Advantages of Data Replication

- ♦ **High availability:** If one node fails, data can still be accessed from other replicas.
- ♦ **Fault tolerance:** Protects against data loss in case of hardware or network failures.
- ♦ **Improved performance:** Reduces latency for read operations by allowing users to access data from the nearest replica.
- ♦ **Load balancing:** Distributes read and write operations across multiple nodes, reducing the load on a single server.
- ♦ **Disaster recovery:** Provides a backup of data in case of catastrophic failures.

6.2.2.6 Disadvantages of Data Replication

- ♦ **Storage overhead:** Maintaining multiple copies of data increases storage requirements.
- ♦ **Consistency issues:** Ensuring consistency across replicas can be challenging, especially in asynchronous replication.
- ♦ **Complexity:** Managing and synchronizing replicas adds complexity to the system.

6.2.2.7 Replication vs. Fragmentation

- ♦ **Replication:** Focuses on creating multiple copies of data for availability and performance.
- ♦ **Fragmentation:** Focuses on dividing data into smaller pieces for manageability and performance.

6.2.3 Allocation Techniques for Distributed Database Design

The allocation strategy itself, the assignment of fragments to sites, can take various forms. Centralized allocation, where the entire database resides at a single site, simplifies management but creates a single point of failure and potential performance bottlenecks. Partitioned allocation, which assigns each fragment to a unique site, minimizes redundancy but can lead to performance issues when data is accessed remotely. Replicated allocation, which stores copies of fragments at multiple sites, improves availability and performance but necessitates mechanisms for maintaining data consistency. The selection of an allocation strategy is influenced by a multitude of factors, including data access patterns, network characteristics, storage capacity, data consistency requirements, availability requirements, and cost considerations. Data allocation refers to the process of deciding where to store data fragments (e.g., tables, rows, or columns) in a distributed database system. It involves determining the optimal placement of data across nodes to achieve performance, availability, and cost goals.

6.2.3.1 Purpose of Data Allocation

◆ Performance Optimization

Placing data closer to where it's most frequently accessed reduces network traffic and query response times.

◆ Availability Enhancement

Distributing data across multiple sites increases fault tolerance, ensuring data remains accessible even if a site fails.

◆ Cost Efficiency

Balancing storage and communication costs by strategically allocating data.

6.2.3.2 Factors affecting data allocation decisions

Selecting an appropriate data allocation strategy depends on several factors, including data access patterns, network bandwidth, reliability requirements, and update frequency. If users are geographically dispersed, partitioning and replication help reduce access latency. If data updates are frequent, centralized or partitioned storage may be preferred to avoid the complexity of synchronizing multiple copies.

Network bandwidth plays a critical role in allocation decisions. If network latency is high, storing data closer to users minimizes delays. On the other hand, if bandwidth is sufficient, replication becomes a viable option to enhance availability. Reliability requirements also influence allocation strategies. In mission-critical applications where data must always be accessible, replication is essential. However, for cost-sensitive applications, partitioning may be more efficient.

Storage costs must also be considered, as maintaining multiple copies of data in a replicated system increases storage requirements. Additionally, load balancing must be managed to prevent excessive queries from overwhelming a particular server. A well-designed distributed database system dynamically adjusts its allocation strategy based on real-time access patterns and resource availability.

6.2.3.3 Data Allocation Strategies

1. Centralized Allocation

Centralized allocation is the simplest form of data distribution, where the entire database is stored at a single site. All users and applications, regardless of their physical location, access the data from this central location. This approach simplifies data management, ensuring strong consistency and eliminating the complexity of synchronizing multiple copies of data. Security policies and access control mechanisms are also easier to enforce since all data resides in one place.

However, centralized allocation has significant drawbacks in a distributed environment. It leads to high network traffic, as every user must retrieve data from the central site, increasing response times for geographically dispersed users. Moreover, it creates a single point of failure—if the central site becomes unavailable due to a system crash or network failure, access to the database is entirely lost. For these reasons, centralized allocation is rarely used in modern distributed database systems, except in small-scale environments where data access is limited to a specific geographical region.

2. Fragmented Allocation

Fragmented allocation in distributed database design involves dividing a database into smaller, independent fragments and distributing them across multiple sites to optimize access and performance. Fragmentation can be horizontal, where rows of a table are divided based on specific criteria and stored at different locations, or vertical, where columns of a table are separated and stored at different sites. A hybrid approach combines both horizontal and vertical fragmentation for greater efficiency. This technique ensures that data is stored closer to the users who access it most frequently, reducing query response time and network traffic. However, fragmented allocation requires careful query optimization and efficient reassembly of data when needed, which can introduce processing overhead. Despite these challenges, fragmentation is widely used in distributed databases to improve scalability, load balancing, and fault tolerance while minimizing unnecessary data transfers.

3. Replicated Allocation

Replication is a technique where copies of the same data are stored at multiple locations to improve availability, reliability, and access speed. It ensures that users can retrieve data from the nearest server, reducing access time and dependency on a single site. Replication can be classified into full replication and partial replication.

Full replication involves storing the entire database at multiple locations, ensuring that every site has a complete copy. This guarantees high availability, as data remains accessible even if some servers fail. It also speeds up query processing by allowing users to access data locally. However, full replication introduces substantial storage and synchronization costs. Maintaining consistency among multiple copies of the database requires frequent updates, leading to increased network traffic and potential data conflicts.

Partial replication, in contrast, involves storing only selected portions of the database at different sites based on demand. This approach balances storage efficiency and

performance by replicating frequently accessed data while keeping rarely used data centralized. For instance, an online retail company may replicate popular product catalogs across multiple regional servers while keeping historical sales data in a central database. Managing partial replication requires careful planning to determine which data should be copied and how often it should be synchronized.

While replication improves fault tolerance, it complicates consistency management. If multiple users update replicated data simultaneously, mechanisms such as synchronous or asynchronous replication must be implemented to ensure data integrity. Synchronous replication updates all copies in real-time, ensuring consistency but increasing response time. Asynchronous replication allows updates to propagate with some delay, improving performance but increasing the risk of data inconsistencies.

4. Hybrid Allocation

Hybrid allocation in distributed database design combines partitioning (fragmentation) and replication to optimize performance, availability, and storage efficiency. In this approach, data is first partitioned—either horizontally, vertically, or both—so that subsets of data are stored at different locations based on access patterns. Frequently accessed partitions are then selectively replicated across multiple sites to improve availability and fault tolerance. This method balances the benefits of both partitioning and replication by ensuring that data is stored close to where it is needed while maintaining redundancy for reliability. However, hybrid allocation introduces complexity in data consistency management, query optimization, and synchronization, as updates must be efficiently propagated across replicated fragments. Despite these challenges, hybrid allocation is widely used in large-scale distributed systems where a combination of data locality, availability, and fault tolerance is required, such as global cloud-based applications and multinational enterprises.

Table 6.2.2 Comparison of data allocation strategies

Allocation Strategy	Performance	Fault Tolerance	Cost	Complexity
Centralized Allocation	Poor for remote users.	Single point of failure	One site to manage, so cost is low.	Easy to implement and maintain.
Fragmented Allocation	Good	Some fault isolation	Optimized storage use	High complexity
Replicated Allocation	Excellent	Data available at multiple sites	High	Consistency management is complex
Hybrid Allocation	Best	High	High	Requires careful planning and management

6.2.3.4 Tools and Techniques for Data Allocation

1. Cost Models

Mathematical models to estimate the cost of different allocation strategies.

2. Optimization Algorithms

Algorithms to find the optimal allocation of data fragments.

3. Simulation Tools

Tools to simulate and evaluate different allocation strategies.

4. Monitoring Systems

Systems to monitor access patterns and system conditions for dynamic allocation.

Recap

- ◆ A distributed database stores data on multiple computers instead of just one.
- ◆ Replication copies data to multiple locations so it is still available if one location fails.
- ◆ Distributed databases are reliable because if one computer fails, others can still work.
- ◆ Horizontal Fragmentation: Dividing rows (tuples) based on selection criteria.
- ◆ Vertical Fragmentation: Dividing columns (attributes) based on usage patterns.
- ◆ Mixed/Hybrid Fragmentation: Combining horizontal and vertical fragmentation.
- ◆ Derived Fragmentation: Fragmenting based on relationships with other tables.
- ◆ Synchronous Replication: Real-time updates to all replicas (strong consistency).
- ◆ Asynchronous Replication: Delayed updates to replicas (better performance).
- ◆ Semi-Synchronous Replication: Combination of both.
- ◆ Centralized Allocation: All data at a single site.
- ◆ Fragmented Allocation: Each fragment at a different site.
- ◆ Replicated Allocation: Copies at multiple sites (full or partial).
- ◆ Hybrid Allocation: Combining fragmented and replicated allocation.
- ◆ Data fragmentation, replication, and allocation are crucial techniques for managing and optimizing distributed databases.

Objective Type Questions

1. What does data fragmentation do?
2. What is a potential drawback of data fragmentation in a distributed system?
3. What is vertical fragmentation based on?
4. Where does horizontal fragmentation split data?
5. What is the main benefit of replication?
6. A customer order database, where order details are created based on existing customer data, would effectively utilize what fragmentation type for optimized queries?
7. Derived fragmentation is a special form of which other fragmentation type?
8. A global social media platform, "ConnectNow," needs to ensure user posts are available even if a data center fails. What data management technique is crucial?
9. What does synchronous replication do?
10. Where is all data stored in centralized allocation?
11. What factor significantly influences data allocation in a distributed database?
12. Which factor is most important when determining the optimal data allocation in a distributed system?

Answers to Objective Type Questions

1. Breaks data into pieces.
2. Columns
3. Reconstruction overhead
4. Rows.
5. Availability.
6. Horizontal (derived)
7. Horizontal
8. Data replication.
9. Updates all copies at once.
10. One place.
11. Query frequency
12. The frequency and patterns of data access

Assignments

1. Imagine a small online store with customers in two cities. How would you use fragmentation or replication to improve their database performance?
2. Briefly describe the difference between synchronous and asynchronous replication?
3. Discuss the main goal of data allocation in a distributed database?
4. How do cloud database services (like Amazon RDS or Google Cloud Spanner) handle data replication and allocation? What challenges do they solve?
5. How can data allocation be used to achieve effective load balancing in a distributed database system? Provide examples of allocation strategies that distribute query load evenly across multiple sites.

Suggested Reading

1. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, (2011) *Database System Concepts*.
2. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). *Database system concepts*.
3. Elmasri, R. (2008). *Fundamentals of database systems*. Pearson Education India.
4. Ceri, S., & Pelagatti, G. (1984). *Distributed databases principles and systems*. McGraw-Hill, Inc..
5. Özsu, M. T., & Valduriez, P. (1999). *Principles of distributed database systems* (Vol. 2). Englewood Cliffs: Prentice Hall.

Reference

1. DeWitt, D., & Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 85-98.
2. Van Steen, M. (2002). *Distributed systems principles and paradigms*. Network, 2(28),
3. Lentz, A., Zaitsev, P., Tkachenko, V., Zawodny, J., Balling, D., & Schwartz, B. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). *Database system concepts* (6th ed.). McGraw-Hill.
4. Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

Unit 3

Big data and Data bases

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ recognize the definition and characteristics of Big Data
- ◆ identify the components of the Hadoop ecosystem
- ◆ list the key features of Apache Spark
- ◆ name the types of NoSQL databases
- ◆ recall the challenges of integrating Big Data with traditional databases

Prerequisites

Data has always been a powerful tool for understanding the world. So far, you have learned how to collect, process, and analyze data to uncover patterns and trends. Whether working with spreadsheets, databases, or visualization tools, you have seen how structured data helps businesses make informed decisions. But as technology advances, data is no longer just numbers and tables. It has grown into something much larger and more complex.

Imagine millions of online transactions happening every second, social media posts generating endless streams of content, and sensors in smart devices continuously collecting information. This explosion of data is both an opportunity and a challenge. Traditional data systems struggle to keep up with the sheer volume, variety, and speed at which data is created. Simply storing and processing data is not enough. We need to tell a story with it. Data storytelling helps transform raw information into meaningful insights that drive action.

Big Data and modern databases play a crucial role in making sense of this vast amount of information. With advanced tools and techniques, we can process real-time data, handle unstructured formats like text and images, and uncover patterns that were previously invisible. Companies use these technologies to detect fraud, predict customer behavior, and even improve healthcare outcomes. Learning about Big Data will open up new possibilities for how you work with information.

As you dive into this journey, you will explore how Big Data is stored, processed, and analyzed. You will learn about powerful technologies like Hadoop, Apache Spark,



and NoSQL databases that are shaping the future of data management. The ability to handle and interpret Big Data is a valuable skill in today's digital world. Get ready to discover how data goes beyond numbers and becomes a key driver of innovation and decision-making!

Keywords

Big data, Hadoop, Apache Spark, Data lake, Data warehouse

Discussion

6.3.1 Introduction to Big Data

Data has become the lifeblood of the digital era. Every day, businesses, governments, and individuals generate enormous amounts of data through social media interactions, online transactions, smart devices, and automated systems. This overwhelming volume of data is what we call Big Data. Traditional database systems struggle to handle this vast amount of information efficiently. Big Data is not just about size; it is also about speed, variety, and the value it provides.

Organizations collect structured data, such as customer transactions, semi-structured data like XML and JSON files, and unstructured data from social media, emails, and multimedia content. Big Data enables industries to make informed decisions, enhance customer experiences, and predict future trends. For example, e-commerce platforms analyze millions of customer interactions daily to recommend personalized products, while healthcare providers examine patient data to predict disease outbreaks.

6.3.2 Traditional Data Vs Big data

Data is everywhere, helping businesses and organizations make smarter decisions. But not all data is the same! Based on how much data there is, how fast it grows, and how it's handled, we can divide it into two main types: Traditional Data and Big Data. Understanding their differences is key to choosing the right tools for storing, processing, and analyzing information.

Think of traditional data as neatly organized information that fits into tables, spreadsheets, or structured databases. This includes things like customer details, sales records, or employee databases. Traditional data is structured and organized, meaning it is stored in fixed formats that make it easy to manage and retrieve. It is typically accessed and manipulated using Structured Query Language (SQL), which allows users to efficiently manage the data within relational databases. Traditional data is widely used in business applications and is often managed by enterprise resource planning (ERP) systems and other enterprise-level software to support decision-making and operational processes.

Big Data is like the bigger, more complex cousin of traditional data. It includes vast amounts of structured, semi-structured, and unstructured data from various sources like social media, IoT sensors, video streams, website clicks, and more! The 5 Vs define big data:

Volume –	Huge amounts of data generated every second.
Velocity –	Data flows in at an incredible speed.
Variety –	Comes in many formats (text, images, videos, etc.).
Veracity –	The reliability of data can vary.
Value –	Extracting meaningful insights is the main goal.

Since traditional databases can't handle such massive and diverse data, specialized tools like Hadoop, Spark, and NoSQL databases are used to process and analyze it.

6.3.3 Big Data Technologies

As already mentioned, advanced tools and frameworks are designed to store, process, and analyze massive volumes of structured, semi-structured, and unstructured data in real time. From social media interactions and IoT sensor data to online transactions and video streams, big data technologies help businesses extract meaningful insights, enhance decision-making, and drive innovation. Popular technologies like Hadoop, Spark, and NoSQL databases enable organizations to manage and analyze data efficiently, unlocking new opportunities in artificial intelligence, machine learning, and predictive analytics.

6.3.3.1 Hadoop

As data grows exponentially, specialized technologies have emerged to process, store, and analyze it effectively. One of the foundational technologies is Hadoop, an open-source framework that distributes data across multiple machines. Hadoop consists of two key components that work together to handle large-scale data:

1. Hadoop Distributed File System (HDFS):
 - ◆ Stores data across multiple machines by breaking it into smaller chunks.
 - ◆ Ensures fault tolerance by maintaining multiple copies of data, so even if a machine fails, the data remains accessible.
2. MapReduce:
 - ◆ A parallel processing model that divides large computations into smaller tasks, distributing them across machines.
 - ◆ After processing, it combines results to generate meaningful insights efficiently.

A global retail company like Amazon handling billions of transactions daily. Analyzing customer purchasing trends across different regions using traditional databases would be slow and inefficient. However, with Hadoop: HDFS stores transaction records across multiple servers. MapReduce processes this data simultaneously, identifying trends like which products are popular in different countries or which promotions drive the most sales. The company can then use these insights to optimize inventory, personalized recommendations, and improve customer experience.

6.3.3.2 Apache Spark

Apache Spark is another powerful Big Data processing tool. Unlike Hadoop's MapReduce, which processes data in batches, Spark enables real-time data analysis. This is crucial for applications like fraud detection in banking, where transactions must be monitored continuously to identify suspicious activities. Spark's ability to process data quickly makes it suitable for stock market analysis, where decisions need to be made within seconds.

Consider a bank that processes millions of transactions daily. Traditional systems using batch processing may take hours or even days to detect fraudulent activities, leaving customers vulnerable to financial loss. But with Apache Spark, every transaction is monitored in real-time, allowing the system to instantly analyze spending patterns and flag unusual activities, such as a sudden large withdrawal from a foreign country. If a suspicious transaction is detected, the bank can immediately block it or alert the customer for verification. By reducing detection time from hours to milliseconds, Spark enhances security and helps prevent fraud before any damage occurs.

6.3.3.3 NoSQL Databases

Traditional relational databases struggle to manage the complexity of Big Data. NoSQL databases provide a flexible alternative. MongoDB, a document-based database, stores data in a JSON-like format, making it ideal for content management systems. Cassandra, a column-family store, handles distributed data efficiently and is used by social media platforms to store user interactions. Neo4j, a graph database, excels at managing highly connected data, such as social networks and recommendation engines. For example, an online streaming service may use Neo4j to recommend movies based on users' viewing history.

6.3.3.4 Cloud-Based Big Data Solutions

Cloud computing has changed how organizations store and process Big Data. It offers flexible and cost-effective solutions. Traditional data storage needs expensive hardware and regular maintenance. Cloud platforms like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure provide on-demand computing power. Businesses can store and manage large amounts of data without needing physical servers. These platforms make it easier to handle structured and unstructured data efficiently.

Each cloud service provides tools to improve data processing. Amazon Redshift is a cloud-based data warehouse that runs complex queries quickly. Google BigQuery processes huge datasets in seconds and uses AI to provide better insights. Azure Synapse Analytics combines data from different sources to help businesses make better decisions. These cloud tools remove the need for expensive infrastructure. They also provide high security and allow data to be processed in real time.

A multinational company can use cloud computing to combine data from all its branches. Instead of keeping separate data centers in each country, it can store everything in one cloud system. This allows real-time reporting and better teamwork across

locations. The company can analyze customer buying trends, track its supply chain, and predict market changes easily. Cloud-based Big Data solutions help businesses make faster and smarter decisions. They also lower costs and improve overall efficiency.

6.3.4 Integration with Database Systems

As businesses generate vast amounts of data, integrating Big Data technologies with traditional database systems has become essential. Modern organizations use a combination of data warehouses, data lakes, real-time processing tools, and hybrid database systems to store, manage, and analyze data efficiently. This integration helps businesses gain deeper insights, improve decision-making, and enhance operational efficiency. By leveraging structured and unstructured data together, companies can unlock new opportunities and optimize their data-driven strategies.

1. Data Warehouses and Data Lakes

A data warehouse is a structured storage system that consolidates data from different sources. Organizations traditionally relied on data warehouses for structured information, but with the rise of Big Data, they now integrate data lakes. A data lake allows raw, unstructured data to be stored alongside structured data. It works by storing vast amounts of raw data from multiple sources in its native format until it is needed for analysis. Data enters the lake through various channels such as databases, IoT devices, social media, and application logs. For instance, a retail company may keep sales transactions in a data warehouse while storing customer reviews, website logs, and social media feedback in a data lake. By combining these datasets, the company can gain a more comprehensive understanding of customer preferences.

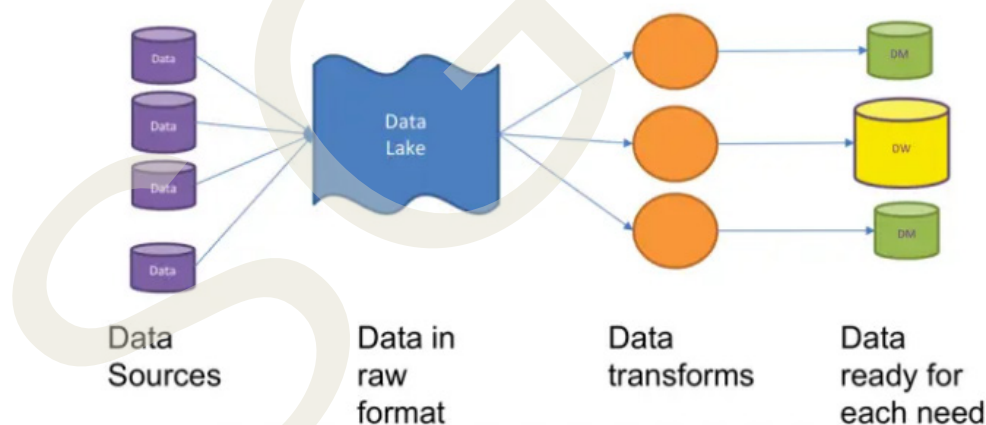


Fig. 6.3.1 Data Lake Pattern

2. Real-Time Data Processing

Real-time data processing has become a necessity for many industries. Apache Kafka and Apache Flink enable organizations to process and analyze streaming data instantly. A stock trading platform, for example, relies on real-time data to monitor fluctuations and execute trades within milliseconds. Similarly, ride-sharing apps like Uber use real-time analytics to match drivers with passengers efficiently and predict demand patterns.

3. Hybrid Database Systems

Many organizations adopt a hybrid approach, combining relational databases with Big Data platforms. A hospital, for instance, may use a relational database to store structured patient records while utilizing a NoSQL database to manage medical images and sensor data from wearable devices. This hybrid integration ensures that different types of data can be accessed and analyzed without performance bottlenecks.

6.3.5 Challenges of Big Data Integration

Integrating Big Data with traditional databases comes with challenges such as data consistency and scalability. Since data is collected from multiple sources at different times, ensuring accuracy and eliminating discrepancies can be difficult. As data volume grows, storage and processing must scale efficiently to maintain performance. Security and privacy are also major concerns, especially when handling sensitive data like financial transactions and health records. Preventing unauthorized access, data breaches, and ensuring compliance with regulations require strong security measures.

Managing the cost and resources of Big Data solutions is another challenge, as organizations require advanced hardware, cloud storage, and skilled professionals. Businesses adopt cloud-based storage, automation, and security protocols to optimize data management. Despite these challenges, Big Data continues to transform industries by improving decision-making, enhancing customer experiences, and driving innovation. With advancements in cloud computing, AI, and real-time analytics, businesses across healthcare, finance, and retail are leveraging Big Data for efficiency and growth. Embracing these technologies will be key to staying competitive in a data-driven world.

Recap

Big Data Overview

- ◆ Massive data generated daily from social media, transactions, IoT, and automation
- ◆ Traditional databases struggle with handling vast, fast-growing data
- ◆ Big Data focuses on size, speed, variety, and value

Traditional Data vs. Big Data

- ◆ Traditional data – Structured, stored in databases, managed using SQL
- ◆ Big Data – Includes structured, semi-structured, and unstructured data
- ◆ 5 Vs of Big Data – Volume, Velocity, Variety, Veracity, Value
- ◆ Specialized tools like Hadoop, Spark, and NoSQL handle Big Data

Big Data Technologies

- ◆ Used for real-time analytics, decision-making, and innovation
- ◆ Common tools – Hadoop, Spark, NoSQL databases, Cloud solutions

Hadoop and Distributed Processing

- ◆ HDFS (Hadoop Distributed File System) – Stores data across multiple machines, ensures fault tolerance
- ◆ MapReduce – Processes large-scale data in parallel
- ◆ Example: Amazon uses Hadoop to analyze global sales trends

Apache Spark

- ◆ Faster than Hadoop, supports real-time processing
- ◆ Used in fraud detection, stock market analysis, ride-sharing apps
- ◆ Example: Banks use Spark to detect fraud instantly

NoSQL Databases

- ◆ Handle unstructured and high-volume data
- ◆ Types: MongoDB (document-based), Cassandra (distributed storage), Neo4j (graph-based)
- ◆ Example: Streaming services use Neo4j for personalized recommendations

Cloud-Based Big Data Solutions

- ◆ Scalable, cost-effective alternative to traditional storage
- ◆ AWS, Google Cloud, Azure provide on-demand storage and computing
- ◆ Amazon Redshift, Google BigQuery, Azure Synapse optimize data processing
- ◆ Example: Multinational companies store data centrally for real-time analysis

Integration with Database Systems

- ◆ Combines structured databases with Big Data tools
- ◆ Includes Data Warehouses, Data Lakes, Real-Time Processing, Hybrid Databases

Data Warehouses vs. Data Lakes

- ◆ Data Warehouse – Stores structured, processed data from different sources
- ◆ Data Lake – Stores raw, unstructured, and structured data together
- ◆ Example: Retailers store sales in warehouses, customer feedback in lakes

Real-Time Data Processing

- ◆ Tools: Apache Kafka, Apache Flink for real-time analysis
- ◆ Used in stock trading, ride-sharing apps, fraud detection

Hybrid Database Systems

- ◆ Combines SQL databases with NoSQL for diverse data needs
- ◆ Example: Hospitals use SQL for patient records, NoSQL for medical images

Challenges of Big Data Integration

- ◆ Data Consistency – Ensuring accuracy across multiple sources
- ◆ Scalability – Expanding storage and processing efficiently
- ◆ Security & Privacy – Protecting sensitive financial and medical data
- ◆ Cost & Resource Management – Requires cloud storage, skilled professionals
- ◆ Despite challenges, Big Data drives efficiency, innovation, and competitive advantage

Objective Type Questions

1. What is the term for the massive amount of data generated daily from various sources?
2. Which traditional query language is used to manage structured data?
3. What is the main storage system used in Hadoop?
4. Which programming model in Hadoop processes data in parallel?
5. What is the key advantage of Apache Spark over Hadoop?
6. Name a real-time data processing tool used in stock market analysis.

7. Which NoSQL database is document-based and stores data in a JSON-like format?
8. What type of database is Cassandra classified as?
9. Which graph-based database is used for recommendation engines?
10. Name a cloud-based data warehouse provided by Amazon.
11. What cloud service by Google is used for processing large datasets?
12. Which cloud platform offers Azure Synapse Analytics?
13. What type of storage system holds both raw unstructured and structured data?
14. Name one real-time data processing tool other than Apache Spark.
15. What is a major challenge in Big Data integration related to sensitive information?

Answers to Objective Type Questions

1. Big Data
2. SQL
3. HDFS
4. MapReduce
5. Real-time processing
6. Apache Spark
7. MongoDB
8. Column-family store
9. Neo4j
10. Amazon Redshift
11. Google BigQuery
12. Microsoft Azure
13. Data Lake
14. Apache Kafka
15. Security and Privacy

Assignments

1. Compare and contrast traditional databases and Big Data technologies. Explain how businesses decide which one to use based on their data needs.
2. Discuss the role of Hadoop in Big Data processing. Explain the functions of HDFS and MapReduce with real-world examples.
3. Explain the advantages of Apache Spark over Hadoop's MapReduce. Provide examples of industries that benefit from real-time data processing.
4. What are NoSQL databases, and why are they important for Big Data? Compare different types of NoSQL databases and their applications.
5. How has cloud computing changed Big Data storage and processing? Discuss the benefits and challenges of using cloud-based Big Data solutions.

Suggested Reading

1. Big Data: Principles and Best Practices of Scalable Real-Time Data Systems – Nathan Marz and James Warren
2. Hadoop: The Definitive Guide – Tom White
3. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence – Pramod J. Sadalage and Martin Fowler
4. Cloud Computing: Concepts, Technology & Architecture – Thomas Erl, Zaigham Mahmood, and Ricardo Puttini

Reference

1. Database Management Systems Raghu Ramakrishnan and Johannes Gehrke, Third Edition, McGraw Hill, 2003
2. Database Systems: Design, Implementation and Management, Peter Rob, Thomson Learning, 7Edn.
3. Concept of Database Management, Pratt, Thomson Learning, 5Edn.
4. Database System Concepts - Silberschatz, Korth and Sudarsan, Fifth Edition, McGraw Hill, 2006
5. The Complete Reference SQL - James R Groff and Paul N Weinberg

Unit 4

Future Trends in Databases

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ familiarize with the concept of cloud databases and its key components
- ◆ define an In-Memory Database and list its key features
- ◆ familiarize DBaaS

Prerequisites

In the digital era, data is the backbone of every business, application, and decision-making process. You have already explored distributed databases and how they help manage large-scale data by distributing it across multiple systems to enhance reliability and performance. You have also studied Big Data, which deals with the enormous volume, variety, and velocity of data generated every second, from social media interactions to IoT sensor readings.

How Has the World Changed?

As technology advances, traditional database systems face new challenges. Managing vast amounts of data with on-premises databases requires expensive hardware, regular maintenance, and complex scalability solutions. This approach worked well when data was relatively structured and predictable. However, today's world demands: Remote accessibility (Applications and users are spread across the globe), Real-time processing (Businesses need instant insights, not delayed reports), Scalability (Companies experience unpredictable traffic spikes e.g., e-commerce sales, live streaming events), Cost efficiency (Maintaining hardware and IT infrastructure is expensive).

This shift in data needs has led to the rise of Cloud Databases, In-Memory Databases, and Database as a Service (DBaaS) offering flexibility, high performance, and cost-effectiveness. Organizations now rely on cloud-based solutions that can dynamically scale and eliminate infrastructure complexities.

Now that businesses and applications demand faster, smarter, and more scalable database solutions, what does the future hold? Emerging trends like AI-powered databases, blockchain-based storage, multi-model databases, and serverless database architectures are redefining data management.



In this next section, you will explore how these cutting-edge database technologies are shaping the future and revolutionizing how we store, retrieve, and analyze data.

Keywords

In-Memory Database, Data Storage, Cloud Database, DBaaS

Discussion

Data plays a vital role in today's digital era, and managing it efficiently is essential for businesses and individuals alike. Traditionally, databases were stored on physical servers, requiring costly hardware, regular maintenance, and security management. However, with the rise of cloud computing, cloud databases have emerged as a more flexible and cost-effective solution.

6.4.1 What is a Cloud database?

A cloud database is a database service hosted and accessed via a cloud computing platform. It functions similarly to a traditional database but offers enhanced scalability, flexibility, and remote accessibility. Users deploy and manage the database on cloud infrastructure without the need for on-premises hardware, enabling efficient storage, retrieval, and management of data.

Managing engagement and application data for large networks of mobile users or remote devices presents challenges in scalability and availability. Traditional databases often rely on a central master database for updates, which can create performance bottlenecks. If the connection to the master database fails, applications may become unresponsive, disrupting operations and user experience. Cloud databases address these challenges by offering distributed architectures that enhance scalability and availability. Unlike traditional databases that rely on a central master, cloud databases distribute data across multiple servers and regions, reducing bottlenecks and minimizing downtime. This ensures continuous operation, even if one node or connection fails, making them ideal for managing large-scale mobile users and remote devices efficiently.

6.4.1.1 Key Components of Cloud database Functionality

Cloud databases function like a traditional database but are hosted on a cloud platform, offering benefits such as scalability, high availability, and reduced management overhead. It eliminates the need for on-premises hardware and infrastructure, as cloud providers handle maintenance, updates, and backups. Following are the key components of cloud databases.

1. Data Storage

Cloud databases store data on virtualized servers within cloud provider data centers. The data is often distributed across multiple servers to ensure redundancy, security, and high availability.

2. Internet-Based Access

Users interact with cloud databases through a web interface, APIs, or SQL queries, enabling remote access from any location with an internet connection.

3. Scalability

Cloud databases offer dynamic scaling, allowing resources to be adjusted based on data volume and usage demands. This flexibility ensures optimal performance without over-provisioning or resource wastage. For example, an e-commerce website may experience moderate traffic on regular days but see a surge during major sales events like Black Friday. During peak periods, the cloud database automatically scales up by increasing computing power and storage to handle the high number of transactions smoothly, preventing slowdowns or crashes. Once the traffic returns to normal, the database scales down, reducing resource consumption and costs. This flexibility ensures optimal performance, cost efficiency, and seamless user experience, making cloud databases ideal for businesses with fluctuating workloads.

4. Managed Services

Cloud providers handle the underlying infrastructure management, including: Hardware maintenance, software updates, security patches and automatic backups. This reduces the operational burden on users and ensures reliable database performance.

5. Data Replication and High Availability

Cloud databases replicate data across multiple geographic locations to enhance data integrity, fault tolerance, and disaster recovery. This ensures that applications remain functional even in case of hardware failures or network disruptions.

6. Support for Multiple Database Models

Cloud databases support a variety of database models, depending on application needs:

- ◆ Relational (SQL databases) : Structured, transactional data storage.
- ◆ NoSQL databases (document, key-value, graph) : Flexible storage for unstructured and semi-structured data.
- ◆ Cloud databases are widely used in modern applications, providing businesses with a reliable, scalable, and cost-effective data management solution. A hybrid cloud integrates public cloud, private cloud, and on-premises infrastructure into a unified, flexible, and cost-effective IT environment.

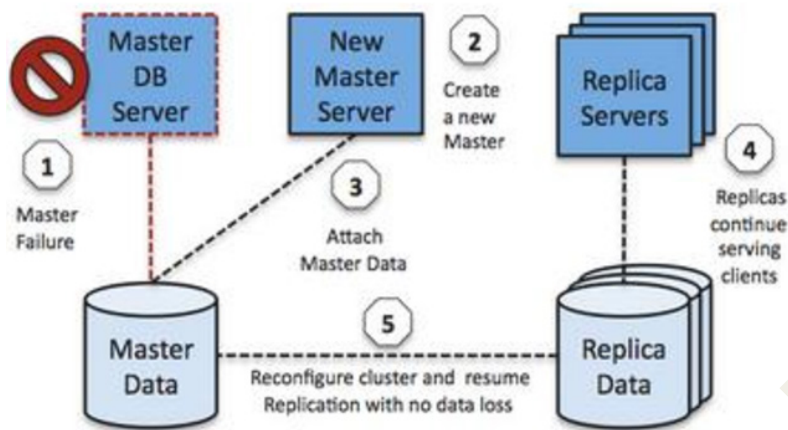


Fig. 6.4.1 Working of Cloud database

6.4.1.2 Benefits of Cloud databases

The benefits of cloud databases are:

- ◆ **Ease of Access:** Cloud databases can be accessed from anywhere using a vendor-provided API or web interface, ensuring seamless remote connectivity.
- ◆ **Scalability:** Cloud databases dynamically adjust their storage capacity in real-time to meet changing demands, allowing organizations to pay only for the resources they use.
- ◆ **Disaster Recovery:** In case of natural disasters, hardware failures, or power outages, cloud databases ensure data security through automated backups on remote servers.

6.4.2 What is In-Memory Databases

An In-Memory Database (IMDB) is a type of database that stores data in the main memory (RAM) instead of traditional disk storage. This architecture enables ultra-fast data access, making IMDBs suitable for applications requiring real-time processing, such as financial transactions, online gaming, and telecommunications. IMDBs remove delays caused by disk access, making queries run much faster and improving system performance. They allow many users to access and update data at the same time without slowing down. Some IMDBs can also save data to disk regularly to prevent data loss if the system crashes. They can easily grow and work across multiple servers to manage large amounts of data efficiently.

6.4.2.1 Why Use an In-Memory Database?

Traditional databases store data on hard disks or SSDs, which require disk I/O (Input/Output) operations every time data is read or written. This process introduces latency (delay) because accessing data from disk takes more time compared to accessing it from RAM.

In contrast, In-Memory Databases (IMDBs) eliminate these delays by storing all data in the system's RAM. Since RAM operates at much higher speeds than disk storage, query execution and data retrieval happen almost instantly, significantly improving efficiency and performance.

6.4.2.2 Features of In-Memory Database

1. High Speed & Low Latency in In-Memory Databases

In-Memory Databases store data in RAM, eliminating disk I/O delays and enabling instant access. This makes them ideal for real-time applications like banking, trading, and AI analytics.

2. Volatility & Persistence Options

RAM is volatile, meaning data is lost when the system is restarted or crashes. Some IMDBs offer persistence mechanisms such as periodic snapshots or transaction logs to store data on disk as a backup.

3. Efficient Memory Management

IMDBs use data compression, indexing, and parallel processing to save memory and speed up queries.

4. Concurrency & Scalability

IMDBs support multiple transactions at the same time with minimal delays. Some IMDBs use distributed memory architecture, allowing them to scale across multiple servers.

6.4.2.3 Examples of In-Memory Databases

Redis : A fast in-memory database that stores key-value data, mainly used for caching and real-time applications.

Memcached : A simple in-memory caching system that helps speed up websites by storing frequently used data.

SAP HANA : A powerful in-memory database that processes large amounts of data quickly for real-time analytics.

Apache Ignite : A distributed in-memory database that speeds up data processing and supports SQL queries.

SQLite (In-Memory Mode) : A small database that runs entirely in RAM, useful for fast, temporary data storage.

6.4.3 Database as a Service (DBaaS)

Data is a crucial asset for IT organizations and businesses of all sizes and industries. Every day, organizations generate vast amounts of data, which drives various operational



activities. However, managing databases and ensuring their security remains a significant challenge. To address this, new technologies are continually emerging in the database domain to enhance efficiency and flexibility. One such advancement is Database as a Service (DBaaS), which simplifies database management. This article explores the concept and benefits of DBaaS.

Database as a Service (DBaaS) is a cloud-based managed database solution that allows users to store, manage, and retrieve data without the need for manual database maintenance, configuration, or infrastructure management. It provides database functionalities as a service, enabling businesses to focus on application development rather than database administration.

Key Features of DBaaS

1. **Fully Managed Service** : The cloud provider handles database setup, configuration, patching, and updates.
2. **Scalability** : Can scale up or down automatically based on workload demand.
3. **High Availability** : Ensures continuous uptime with replication and failover mechanisms.
4. **Backup and Recovery** : Automated backups and disaster recovery options are included.
5. **Security and Compliance** : Provides encryption, access control, and compliance with industry standards.
6. **Multi-Cloud and Hybrid Support** : Some DBaaS solutions work across different cloud platforms or hybrid environments.
7. **Performance Optimization** : Offers automated indexing, caching, and query tuning.

Advantages of DBaaS

- ◆ **Reduced Operational Overhead** : No need to manage hardware or software manually.
- ◆ **Cost Efficiency** : Pay-as-you-go pricing models reduce upfront costs.
- ◆ **Faster Deployment** : Quick provisioning of databases without complex setup.
- ◆ **Automatic Maintenance** : Updates and patches are handled by the provider.
- ◆ **Improved Security** : Built-in encryption and access control mechanisms.

Common DBaaS Providers

1. **Amazon RDS** (Relational Database Service) : Supports MySQL, PostgreSQL, SQL Server, Oracle, etc.

2. **Google Cloud SQL** : Fully managed database for MySQL, PostgreSQL, and SQL Server.
3. **Microsoft Azure SQL Database** : Cloud-based SQL database with AI-driven performance optimization.
4. **IBM Cloud Databases** : Supports multiple databases like MongoDB, PostgreSQL, and Redis.
5. **MongoDB Atlas** : Fully managed NoSQL database service for MongoDB.

Use Cases of DBaaS

- ◆ Web applications needing scalable and reliable databases.
- ◆ Enterprises looking to reduce database management workload.
- ◆ Startups requiring cost-effective and quick database solutions.
- ◆ Data analytics platforms that require high-speed data processing.

Recap

- ◆ Cloud database is hosted on a cloud platform
- ◆ No need for on-premises hardware
- ◆ Enables efficient data storage, retrieval, and management.
- ◆ Traditional databases rely on a central master, causing bottlenecks and downtime risks.
- ◆ Cloud databases distribute data across multiple servers for better availability.
- ◆ Ensures continuous operation even if a node or connection fails.

Key Components of Cloud Database Functionality

- ◆ **Data Storage** : Stored on virtualized servers, ensuring redundancy, security, and availability.
- ◆ **Internet Access** : Users connect via web interface, APIs, or SQL from anywhere.
- ◆ **Scalability** : Resources scale dynamically based on demand, optimizing cost and performance.
- ◆ **Managed Services** : Cloud providers handle maintenance, updates, security, and backups.
- ◆ **Data Replication** : Data copies stored across locations for reliability and disaster recovery.

- ◆ Database Models
 - SQL databases
 - NoSQL databases

Benefits of Cloud Databases

- ◆ Ease of Access : Accessible via API or web interface from anywhere.
- ◆ Scalability : Adjusts storage in real-time based on demand.
- ◆ Disaster Recovery : Automated backups protect data from failures.

In-Memory Database (IMDB) stores data in RAM instead of disk for ultra-fast access.

Key Features of In-Memory Database

- ◆ High Speed & Low Latency : Ideal for real-time applications like banking, trading, and AI analytics.
- ◆ Volatility & Persistence : Data loss risk, but some IMDBs support periodic backups.
- ◆ Efficient Memory Management : Uses compression, indexing, and parallel processing for optimization.
- ◆ Concurrency & Scalability : Supports multiple transactions and can scale across multiple servers.

Key Points on Database as a Service (DBaaS)

- ◆ Cloud-based managed database solution
- ◆ Eliminates manual maintenance, configuration, and infrastructure management
- ◆ Enables businesses to focus on application development

Key Features of DBaaS

- ◆ Fully Managed : Cloud provider handles setup, updates, and patching
- ◆ Scalability : Adjusts resources based on workload demand
- ◆ High Availability : Ensures uptime with replication and failover mechanisms
- ◆ Backup & Recovery : Automated backups for disaster recovery
- ◆ Security & Compliance : Encryption, access control, and regulatory compliance

- ◆ Multi-Cloud & Hybrid Support : Works across different cloud environments
- ◆ Performance Optimization : Automated indexing, caching, and query tuning

Advantages of DBaaS

- ◆ Reduced Operational Overhead : No need for in-house database management
- ◆ Cost Efficiency : Pay-as-you-go pricing minimizes costs
- ◆ Faster Deployment : Quick provisioning without complex setup
- ◆ Automatic Maintenance : Updates and patches managed by provider
- ◆ Improved Security : Built-in encryption and access control
- ◆ Examples of IMDBs : Redis, Memcached, SAP HANA, Apache Ignite, and SQLite (In-Memory Mode).

Objective Type Questions

1. Where is a cloud database hosted?
2. What allows cloud databases to adjust resources based on demand?
3. What do users use to access cloud databases remotely?
4. What stores copies of data across multiple locations?
5. What type of databases offer flexible storage for unstructured data?
6. Where does an In-Memory Database (IMDB) store data?
7. What is eliminated in IMDBs to achieve faster data access?
8. Which type of backup mechanism do some IMDBs use to prevent data loss?
9. Name one example of an In-Memory Database.
10. Which in-memory database is commonly used for caching frequently accessed data in web applications?
11. Which cloud-based service eliminates the need for manual database maintenance?
12. What feature of DBaaS ensures continuous uptime with replication and failover?
13. Name one DBaaS provider that supports MySQL, PostgreSQL, and SQL Server.

14. What pricing model does DBaaS typically follow?
15. What is a major challenge of DBaaS that makes switching providers difficult?

Answers to Objective Type Questions

1. Cloud
2. Scalability
3. API
4. Replication
5. NoSQL
6. RAM
7. Disk I/O
8. Snapshots
9. Redis
10. Memcached
11. DBaaS
12. High Availability
13. Amazon RDS
14. Pay-as-you-go
15. Vendor Lock-in

Assignments

1. Explain the concept of In-Memory Databases (IMDB) and how they differ from traditional disk-based databases.
2. List and describe any three features of In-Memory Databases with examples.

Suggested Reading

1. Designing Data-Intensive Applications by Martin Kleppmann, O'Reilly Media, 2017
2. <https://aws.amazon.com/nosql/in-memory/>

Reference

1. Big Data: Principles and Best Practices of Scalable Real-Time Data Systems" By Nathan Marz and James Warren
2. Hadoop: The Definitive Guide" By Tom White

MODEL QUESTION PAPER SETS



SREENARAYANAGURU OPEN UNIVERSITY

MODEL QUESTION PAPER (SET 1)

QP CODE:

Reg. No:.....

Name:

BACHELOR OF COMPUTER APPLICATIONS

End Semester Examination

BSc.Data Science and Analytics

B24DS04DC- DATABASE MANAGEMENT SYSTEMS

Time: 3 Hours

Max Marks: 70

Section A

Answer any 10 questions. Each carries one mark

(10x1=10)

1. What is data independence?
2. Name a common record-based data model.
3. Define a candidate key.
4. What is a multivalued attribute?
5. Mention one symbol used in ER diagrams.
6. What is the role of a buffer manager?
7. What is an external schema?
8. What is a super key?
9. Give an example of a domain constraint.
10. What is a serial schedule?
11. What does NoSQL stand for?
12. What is a schedule in concurrency control?
13. What is meant by cascading rollback?
14. Name one example of an In-Memory Database.
15. What is the purpose of BCNF?



Section B

Answer any 5 questions. Each carries two marks

(5x2=10)

16. What is the purpose of normalization?
17. Explain the term entity set.
18. What is a composite key? Give an example.
19. Explain partial participation with an example.
20. What is the purpose of recovery manager?
21. What is a relational algebra expression?
22. List two advantages of normalization.
23. Describe Neo4j database.
24. List two reasons why schedules may conflict.
25. What is the key advantage of Apache Spark over Hadoop?

Section C

Answer any 5 questions. Each carries four marks

(5x4=20)

26. Explain the steps to convert ER model to relational model.
27. Describe 1NF, 2NF, and 3NF with examples.
28. Describe two-phase locking protocol.
29. Explain timestamp-based concurrency control.
30. Discuss the advantages of DBMS over traditional file systems.
31. Differentiate between hierarchical and network models.
32. Explain the three levels of data abstraction.
33. Compare one-tier, two-tier, and three-tier architecture.
34. Explain the components of an ER model.
35. Describe specialization and aggregation with examples.

Section D

Answer any 2 questions. Each carries fifteen mark

(2x15=30)

36. With neat diagrams, explain different types of database architectures.
37. Draw and explain ER diagram for a hospital management system.
38. Explain the key components of Cloud database functionality.
39. Explain about transaction states, ACID properties, and concurrency control mechanisms.



SREENARAYANAGURU OPEN UNIVERSITY

MODEL QUESTION PAPER (SET 2)

QP CODE:

Reg. No:.....

Name:

BACHELOR OF COMPUTER APPLICATIONS

End Semester Examination

BSc.Data Science and Analytics

B24DS04DC- DATABASE MANAGEMENT SYSTEMS

Time: 3 Hours

Max Marks: 70

Section A

Answer any 10 questions. Each carries one mark

(10x1=10)

1. Where does an In-Memory Database (IMDB) store data?
2. Define metadata.
3. What does ER stand for in DBMS?
4. Give an example of a composite attribute.
5. What is a weak entity?
6. What is the function of a file manager in DBMS?
7. Define physical schema.
8. Name one example of a derived attribute.
9. What is a tuple in a relation?
10. What is the process of converting ER to relational schema called?
11. Which key can have NULL values in a DBMS?
12. What is a join dependency?
13. Which NoSQL database is suitable for analyzing relationships in IoT networks?
14. What is a transaction?
15. Define atomicity.



Section B

Answer any 5 questions. Each carries two marks

(5x2=10)

16. Compare schema and instance.
17. List any two types of attributes with examples.
18. What is HBase?
19. Explain the role of transaction manager.
20. What are the benefits of using a three-tier architecture?
21. What is a relational schema?
22. What is arity and cardinality in relational models?
23. Distinguish between 3NF and BCNF.
24. Explain conflict serializability with an example.
25. What are the ACID properties?

Section C

Answer any 5 questions. Each carries four marks

(5x4=20)

26. Differentiate between hierarchical and network models.
27. Explain the three levels of data abstraction.
28. Compare one-tier, two-tier, and three-tier architecture.
29. Explain the components of an ER model.
30. Describe specialization and aggregation with examples.
31. Explain the process of converting ER diagrams to relational schema.
32. Discuss the concept of multivalued dependency and 4NF.
33. Explain the key features of MongoDB.
34. Explain query execution plan with an example.
35. What is concurrency control? Explain its need.

Section D

Answer any 2 questions. Each carries fifteen mark

(2x15=30)

36. Design an ER diagram for a university database with at least four entities and relationships.
37. Explain how DBMS components like transaction manager and recovery manager maintain consistency.
38. Discuss the key components of Cassandra database. Describe the various data types and its operations.
39. Compare hierarchical, network, and relational data models with advantages and limitations.

SGOU

സർവ്വകലാശാലാഗീതം

വിദ്യായാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കുതിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുറിപ്പ് ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode

Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery

Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura

Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi

Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

NO TO DRUGS തിരിച്ചിറങ്ങാൻ പ്രയാസമാണ്



ആരോഗ്യ കുടുംബക്ഷേമ വകുപ്പ്, കേരള സർക്കാർ

Database Management System

COURSE CODE: B24DS04DC



Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-989642-9-8



9 788198 964298