SGOU
Higher Education for All

DATABASE TECHNOLOGIES

COURSE CODE: M25CA05DC
Master of Computer Applications
Discipline Core Course
Self Learning Material



SGOU

SREENARAYANAGURU OPEN UNIVERSITY

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

SREENARAYANAGURU OPEN UNIVERSITY

SGOU

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 Technologies

Course Code: M25CA05DC

Semester - II

Discipline Core Course
Postgraduate Programme
Master of Computer Applications
Self Learning Material

SGOU



SREENARAYANAGURU OPEN UNIVERSITY

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

SREENARAYANAGURU
OPEN UNIVERSITY

DATABASE TECHNOLOGIES

Course Code: M25CA05DC
Semester- II
Discipline Core Course
Master of Computer Applications

Academic Committee

Prof. (Dr.) Sabu M.K.
Dr. G. Santhoshkumar T
Prof. (Dr.) Vinod Chandra S.S.
Dr. Vinod P.
Dr.Lajish V.L.
Sreekanth M.S.
Dr. Vivek P.
Dr. Arun K.S.
Dr. Abdul Jebbar P.

Development of the Content

Shamin S.
Greeshma P.P.
Sreerekha V. K.
Dr. Kanitha D.K.
Subi Priya Laxmi S.B.N.
Aswathy V.S.

Review and Edit

Dr. Leena Silvoster M.

Scrutiny

Shamin S.
Greeshma P.P.
Sreerekha V. K.
Dr. Kanitha D.K.
Subi Priya Laxmi S.B.N.
Aswathy V.S.

Cover Design

Jobin J.

Co-ordination

HoS, School of ICS

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

Edition
September 2026

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-181574-1-0

9 788118 157410

SGOU

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.in

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear Learner,

It gives me immense pleasure and a deep sense of pride to warmly welcome you to Sreenarayanaguru Open University—a vibrant and progressive institution committed to transforming lives through inclusive, flexible, and high-quality education.

Established in September 2020 as a forward-looking initiative of the Government of Kerala, the University stands as a beacon of opportunity for learners seeking to advance their academic and professional aspirations through the open and distance learning mode. Guided by our foundational principle that "access and quality define equity," we are steadfast in our mission to democratize education while upholding uncompromising academic standards.

Our university is inspired by the timeless vision and philosophy of Sree Narayana Guru, whose ideals of knowledge, equality, and social transformation continue to guide our academic journey. His enduring legacy instils in us the responsibility to create an educational environment that empowers individuals, nurtures critical thinking, and contributes meaningfully to society.

Understanding the dynamic needs of contemporary learners, we have adopted a robust and learner-centric blended learning model, seamlessly integrating Self-Learning Materials, Academic Counselling, and Advanced Digital Learning Platforms. This holistic approach ensures flexibility without sacrificing academic depth, enabling you to learn at your own pace while remaining meaningfully connected to a vibrant academic ecosystem.

The Master of Computer Applications (MCA) programme you are embarking upon is carefully designed to position you at the forefront of the digital revolution. It uniquely blends strong theoretical foundations with practical, industry-oriented competencies. The curriculum emphasizes algorithmic thinking, system design, programming, database management, networking, and emerging technologies such as artificial intelligence, data science, and cloud computing.
What sets this programme apart is its forward-thinking design, which offers:

- Opportunities for skill enhancement aligned with industry needs
- Multidisciplinary learning pathways for broader intellectual development
- Exposure to innovative and emerging technology domains
- Multiple specialization options tailored to evolving career landscapes
- A strong focus on employability, entrepreneurship, and global career readiness
-

Our Self-Learning Materials are meticulously developed by experts, enriched with contemporary case studies, real-world applications, and practical insights to ensure clarity, engagement, and relevance. We are committed to equipping you not just with knowledge, but with the confidence and competence to excel in a competitive global environment.

At Sreenarayanaguru Open University, you are never alone in your learning journey. Our dedicated learner support system is designed to provide continuous academic guidance, timely assistance, and effective grievance redressal. We encourage you to actively engage with us, share your concerns, and make the most of the resources and support available to you.

As you begin this important phase of your academic journey, I urge you to embrace learning with curiosity, discipline, and determination. The world of technology is ever-evolving, and your willingness to adapt, innovate, and grow will define your success.

Remember, this is not just a programme—it is a pathway to transforming your future.
I wish you a fulfilling learning experience and a successful, inspiring career ahead.

Warm regards,



Prof. (Dr.) Jagathy Raj V. P.
Vice Chancellor

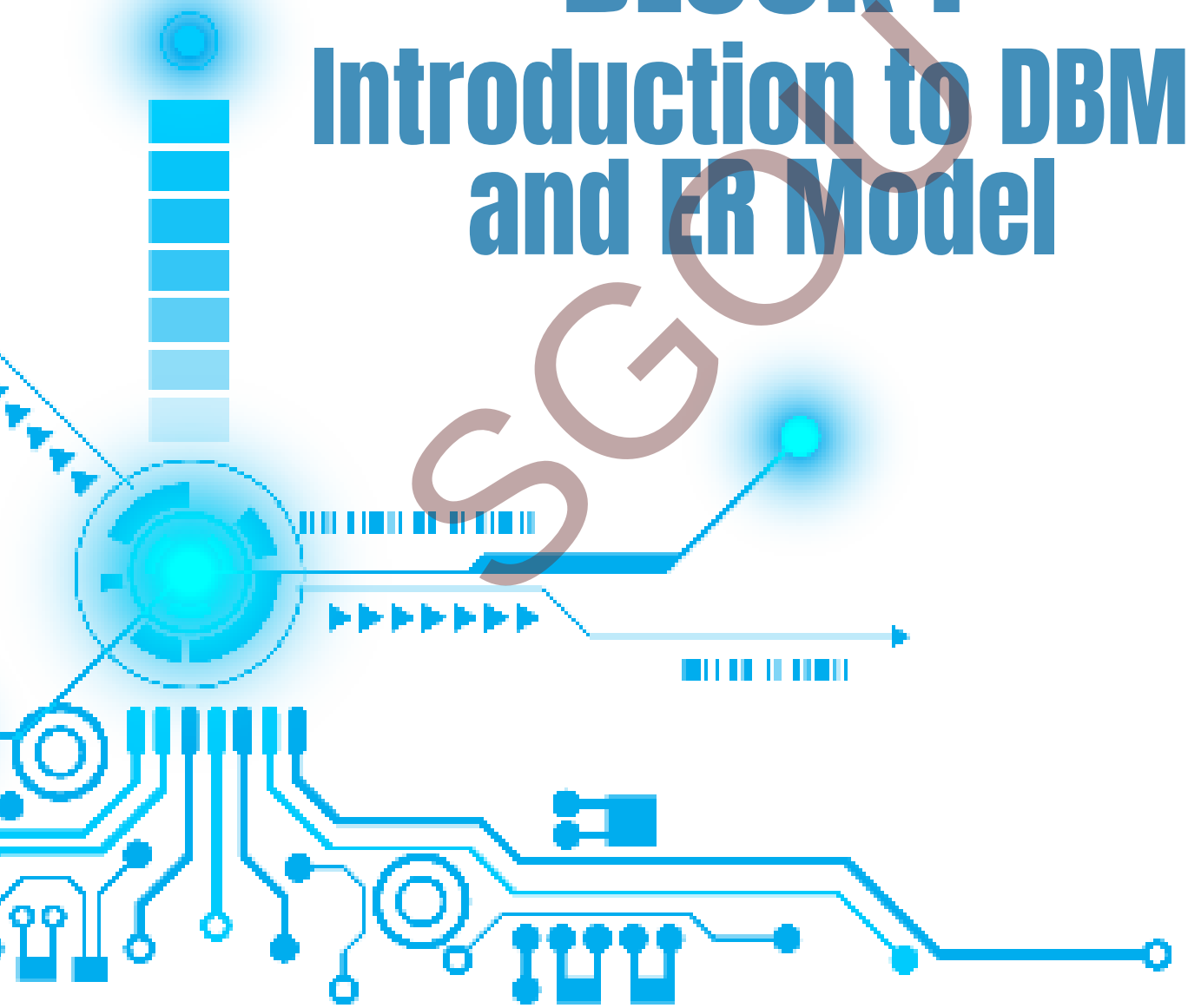
SGOU

Contents

BLOCK 1	**Introduction to DBMS and ER Model**	**1**
UNIT1	Introduction to DBMS	2
UNIT2	Data Models and Schemas	26
UNIT3	ER Model	52
UNIT4	ER Model to Database Schema	67
BLOCK 2	**Relational Database Design and SQL**	**83**
UNIT1	Database Normalization	84
UNIT2	SQL Concepts	109
UNIT3	Built in Functions	136
UNIT4	Views	165
BLOCK 3	**Introduction to PL/SQL**	**182**
UNIT1	Introduction to PL/SQL	183
UNIT2	Cursors	216
UNIT3	Stored Procedures and Functions	235
UNIT4	Triggers	253
BLOCK 4	**Advanced DBMS**	**279**
UNIT1	NoSQL Databases	280
UNIT2	Vector Databases	308
UNIT3	Cloud Databases	335
UNIT4	Multi-Model Databases	358
	Model Question Paper Sets	378

SGOU

BLOCK 1
Introduction to DBMS and ER Model



SGOU

1 UNIT

Introduction to DBMS

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the characteristics, types, and components of Database Management System (DBMS)
- describe the roles and responsibilities of different database users
- differentiate structured, semi-structured, and unstructured data
- analyse the basic principles of database design
- compare different database languages and their purposes

SGOU

Background

Modern organisations generate and manage large volumes of data using applications such as banking, healthcare, e-commerce, education, transportation, and reservation systems. Managing such data using separate files or spreadsheets becomes difficult as the volume of data and number of users increase. Problems such as data redundancy, data inconsistency, difficulties in data sharing, security issues, concurrent access, and data loss may arise.

A Database Management System (DBMS)provides a systematic environment for storing, organising, retrieving, updating, and protecting data. Understanding the basic concepts of databases and DBMS is therefore essential before studying advanced database concepts. This section introduces the fundamental characteristics and types of DBMS, its major components and users, different forms of data, database design stages, and the major categories of database languages.

Keywords

Database Management System (DBMS), Database, Data Integrity, Data Independence, Relational Database, NoSQL Database, Database Design, SQL

Discussion

1.1.1 Introduction to Database Management Systems

Data has become an essential resource in almost every modern organisation. Banks maintain customer accounts and transaction records, hospitals manage patient information, online shopping platforms store product and order details, and transportation systems maintain information about passengers, routes, and reservations. In each of these applications, large quantities of data must be stored, accessed, updated, and protected.

Managing a small volume of data is relatively simple. A few records can be maintained using ordinary files or spreadsheets. However, as the volume of data increases and the number of users grows, several challenges arise. Data redundancy may occur, conflicting values may appear, retrieving information may become difficult, and controlling access to sensitive information may become complicated.

Consider an online shopping platform that maintains information about customers, products, orders, payments, and deliveries. If customer details are stored separately in several files, a change in a customer's address may have to be made in multiple locations. If one instance is not updated, different parts of the system may contain different addresses for the same customer. Such situations demonstrate the need for systematic data management.

A database provides an organised way of storing related data, while a Database Management System provides the software facilities required to manage that data.

1.1.1.1 Data, Information and Database

Before understanding database management concepts, it is important to distinguish between **data** and **information**.

Data refers to raw facts, values, observations, or measurements that can be recorded and processed. A customer's identification number, name, order number, product price, and payment amount are examples of data.

Consider the following details from an online shopping system:

SGOU

Order ID	Customer	Product	Amount
5001	Anu	Laptop	₹65,000

The individual values constitute data. When these values are organised and interpreted together, they provide meaningful information about a particular order.

Information is data that has been processed, organised, or interpreted so that it becomes meaningful and useful for a particular purpose.

For example, the statement *"Order 5001 was placed by Anu for a laptop costing ₹65,000"* represents information derived from the stored data.

The distinction becomes more important as the volume of data increases. An e-commerce company may process thousands of orders every hour. Merely storing these orders is not enough. The organisation may need to determine which products are selling most, which orders are pending, how much revenue was generated during a particular period, or which customers have placed repeated orders.

Therefore, effective data management is not simply about storing data. The data must be organised in a way that allows meaningful information to be obtained when required.

A **database** is an organised collection of related data that can be stored, accessed, and updated systematically.

The word *related* is significant. A database is not simply a collection of unrelated values. Data is organised according to the requirements of a particular application or organisation.

For example, an online shopping database may contain information about customers, products, orders, payments, and deliveries. These data elements are related. An order is associated with a particular customer and one or more products. A payment is associated with an order, while delivery information is associated with the corresponding order.

Such relationships allow the database to represent a meaningful view of the real-world system.

1.1.1.2 Need for Database Management

The need for systematic database management becomes clearer when the volume of data and the number of users increase.

Consider a hospital that maintains patient information, appointment records, laboratory reports, prescriptions, and billing details. If each department maintains separate files containing overlapping patient information, the same data may be stored repeatedly. This creates data redundancy, which refers to unnecessary duplication of data.

Redundancy becomes particularly problematic when information changes. Suppose a patient changes the contact number. The new number may need to be updated in the registration, billing, and appointment records. If one department continues to use the old number, the system contains inconsistent information.

SGOU

This situation is known as data inconsistency, where different copies of supposedly identical data contain conflicting values.

A file-based approach can create several other difficulties.

Finding information may become complicated when relevant data is distributed across multiple files. For example, generating a complete report about a patient's appointments, test results, and payments may require information from several separate files.

Data sharing can also be difficult. Different applications may maintain their own copies of the same information rather than accessing a common, controlled source of data.

Security presents another challenge. Patient information is sensitive and should not be accessible to every employee. Different users may require different levels of access depending on their responsibilities.

Concurrent access is another important concern. A hospital may have reception staff registering patients while doctors retrieve medical records and the billing department processes payments. When several users access the same data simultaneously, the system must ensure that their operations do not produce incorrect or inconsistent results.

Data protection and recovery are equally important. Hardware failures, software errors, accidental deletion, or other unexpected events can result in the loss of valuable information. A reliable data management environment must therefore provide appropriate mechanisms for backup and recovery.

These challenges show why simple file storage becomes inadequate for large and complex applications. A systematic approach is required to organise data, control access, reduce unnecessary duplication, maintain consistency, support multiple users, and protect data from failures.

This requirement is addressed by a Database Management System (DBMS).

1.1.1.3 Database Management System and Database System

A Database Management System (DBMS) is software that enables users to define, create, store, retrieve, update, and manage data within a database.

A DBMS provides an organised environment through which users and application programs can interact with stored data. Users generally do not need to know how individual records are physically stored on storage devices. The DBMS manages these details and provides appropriate mechanisms for accessing the required data.

Consider a banking application. When a customer checks the account balance through a mobile banking application, the customer interacts with the application rather than directly accessing the stored account records. The application communicates with the DBMS, which processes the request and retrieves the required information from the database. The result is then displayed to the customer.

The DBMS performs several important functions, including:

SGOU

- defining the structure of the database
- storing and modifying data
- retrieving required information
- controlling access to data
- maintaining data integrity
- supporting concurrent access
- providing backup and recovery facilities

It is important to distinguish between a database and a DBMS.

A database is an organised collection of related data. A DBMS is the software used to manage that data.

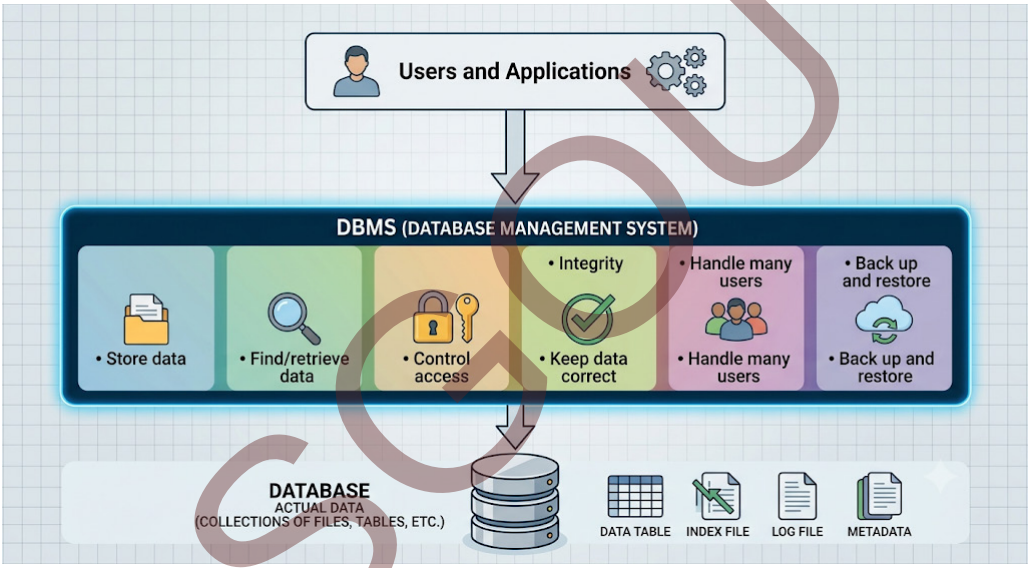


Fig. 1.1.1 DBMS Core Functions

For example, customer accounts and transaction records maintained by a bank constitute a database. The software that manages these records, processes queries, controls access, and maintains the database is the DBMS.

The term database system has a broader meaning. It refers to the complete environment associated with database management. It includes the database, DBMS software, application programs, users, hardware, and procedures.

Different users interact with this environment for different purposes. A customer may retrieve account information, a bank employee may process transactions, an application programmer may develop software that accesses the database, and a database administrator may manage security and recovery.

The growing variety of applications has also resulted in different approaches to database management. Some systems organise data in tables, while others use hierarchical structures, networks, objects, documents, key-value pairs, or graphs.

Thus, a DBMS is not merely a tool for storing data. It provides a systematic environment for managing data and making it available to authorised users and applications in a reliable manner.

1.1.2 Characteristics and types of DBMS

A DBMS is more than a software application used to store records. It provides a collection of features that help organisations manage data accurately, securely, and efficiently. These features distinguish database systems from simple file-based approaches.

At the same time, not all databases organise information in the same way. The nature of the data, the relationships among data items, and the requirements of an application influence the type of DBMS selected.

The characteristics of a DBMS therefore help us understand what a database system provides, while the types of DBMS help us understand the different approaches used to organise and manage data.

1.1.2.1 Characteristics of DBMS

Data Organisation

One of the fundamental characteristics of a DBMS is its ability to organise data systematically.

Consider an online shopping application. Information about customers, products, orders, payments, and deliveries is not stored as a random collection of values. Data is organised in meaningful structures so that the relationships among different items can be represented and relevant information can be retrieved efficiently.

In a relational database, for example, data is commonly organised into tables containing rows and columns. Other database models use different structures.

Good organisation makes it easier to store, retrieve, update, and maintain data.

Data Sharing

A database can be shared by multiple authorised users and applications.

Consider a railway reservation system. Reservation staff, ticketing applications, customer-service systems, and online booking applications may need access to information about trains, passengers, and seats. A common database allows these different users and applications to work with the required information.

However, data sharing does not mean that every user should have unrestricted access. Access must be controlled according to the user's role and requirements.

SGOU

Data Security

Database systems often contain sensitive information. Customer details, financial transactions, medical records, and personal information should not be accessible to unauthorized users.

A DBMS provides mechanisms for authentication and authorisation.

Authentication determines whether a user is who they claim to be. Authorisation determines what that authenticated user is permitted to do.

For example, in a banking system, a customer may be allowed to view their account information, while a bank employee with appropriate privileges may perform additional operations. Different users can therefore be provided with different levels of access.

Data Integrity

A database must contain correct and valid data. Data integrity refers to maintaining the accuracy, consistency, and validity of stored data.

A DBMS can enforce rules called integrity constraints to prevent invalid data from being stored.

For example, a product price should not normally be entered as a negative value. Similarly, a primary key value should not be duplicated across records.

Maintaining integrity becomes particularly important when large numbers of users and applications modify the same database.

Data Redundancy

Data redundancy refers to unnecessary duplication of data.

A well-designed database attempts to reduce unnecessary redundancy. If the same customer details are repeatedly stored in several locations, changes to that information may need to be made multiple times. Reducing unnecessary duplication helps minimise such problems.

However, redundancy is not always completely undesirable. In some database systems, controlled redundancy may be introduced deliberately to improve performance or support particular requirements. Therefore, the objective is not simply to eliminate every duplicate value but to control unnecessary redundancy.

Data Independence

One of the important advantages of a DBMS is data independence.

Data independence refers to the ability to modify the database schema at one level without affecting the schema at another level.

For example, a change in the physical storage details should ideally not require application programs to be rewritten. This separation allows database administrators to make certain internal changes while applications continue to operate.

SGOU

Data independence is an important concept because it reduces the dependence of application programs on the physical details of data storage.

Concurrent Control

Modern database systems often have many users accessing the same data simultaneously.

Consider an online ticket-booking system. Two customers may attempt to reserve the same seat at almost the same time. The DBMS must coordinate these operations so that the same seat is not incorrectly assigned to both customers.

This ability to manage simultaneous operations while preserving data consistency is known as concurrency control.

Concurrency is particularly important in banking, e-commerce, reservation systems, and other applications where many users access shared data.

Backup and Recovery

Data may be lost or damaged because of hardware failures, software errors, power failures, accidental deletion, or other unexpected events.

A DBMS provides facilities that support backup and recovery.

A backup provides a copy of data that can be used when necessary. Recovery involves restoring the database to an appropriate consistent state after a failure.

For an online banking system, for example, losing transaction records could have serious consequences. Therefore, reliable backup and recovery mechanisms are essential.

Data Consistency

Data consistency means that the database remains in a valid and logically correct state.

Suppose an amount is transferred from one bank account to another. The amount should be deducted from one account and appropriately added to the other. If only one operation is completed because of a failure, the database may become inconsistent.

A DBMS uses transaction management mechanisms to help maintain consistency during such operations.

These characteristics show why database management systems are essential for modern applications. However, the way these characteristics are implemented depends partly on the underlying type of DBMS.

1.1.2.2 Types of DBMS

DBMSs can be classified based on the approach they use to organise and manage data. Different types of DBMS have evolved to address different requirements related to data organisation, relationships, flexibility, scalability, and application needs.

SGOU

The major types include hierarchical DBMS, network DBMS, relational DBMS, object-oriented DBMS, object-relational DBMS, and NoSQL database systems.

1. Hierarchical DBMS

A Hierarchical Database Management System organises and manages data using a tree-like structure based on parent-child relationships. It is suitable for applications in which data has a naturally hierarchical organisation.

For example, an organisation with several branches may have departments under each branch and employees under each department. Such an arrangement can be managed using a hierarchical DBMS.

However, applications involving complex relationships that do not follow a strict hierarchy can be difficult to manage using this approach.

2. Network DBMS

A Network Database Management System manages data through interconnected records and supports relationships in which a record can be associated with multiple other records.

For example, in a project management system, an employee may work on several projects, while each project may involve several employees. A network-oriented DBMS can support such complex interconnections.

The network approach provides greater flexibility in representing relationships than a strict hierarchical arrangement.

3. Relational DBMS

A Relational Database Management System (RDBMS) manages data using the relational approach, in which information is commonly organised in tables. It provides facilities for storing, retrieving, updating, and managing structured data and for establishing relationships among related data.

RDBMSs are widely used in banking, retail, healthcare, business applications, and many other domains because they provide a structured environment for managing large collections of related data.

Examples of relational database systems include MySQL, PostgreSQL, Oracle Database, and Microsoft SQL Server.

The detailed concepts associated with the relational approach, including tables, keys, relationships, and other features, will be discussed later while studying data models.

4. Object-Oriented DBMS

An Object-Oriented Database Management System (OODBMS) stores and manages information using object-oriented concepts such as objects, classes, inheritance, and encapsulation.

SGOU

This type of DBMS can be useful for applications that work with complex objects and are closely integrated with object-oriented programming environments.

For example, a computer-aided design system may need to manage complex graphical objects containing multiple properties and relationships.

5. Object-Relational DBMS

An Object-Relational Database Management System (ORDBMS) combines the relational approach with selected object-oriented capabilities. It retains the table-based foundation of relational databases while supporting complex data types and structures.

This approach can be useful when an application requires the established capabilities of relational databases together with support for more complex forms of data.

6. NoSQL Database Systems

NoSQL database systems provide alternatives approaches to the traditional relational model and are designed to address requirements such as flexible data structures, large-scale distributed processing, and high scalability.

Common categories of NoSQL database systems include key-value, document, column-family, and graph database systems. These categories differ in the way they organise and manage data.

For example, a document-oriented database can be useful when application records have flexible structures, while a graph-oriented database can be useful when relationships among data items are particularly important.

The different approaches used to represent and organise data in these database systems will be examined in greater detail while studying data models.

The choice of a DBMS depends on the nature of the application rather than simply on the popularity of a particular technology.

For example, a banking application dealing with highly structured financial transactions may benefit from a relational database. A social networking application that focuses heavily on relationships among users may benefit from graph-oriented databases for specific relationship-intensiveworkloads. An application handling rapidly changing document structures may find a document-oriented database suitable.

Therefore, understanding the characteristics and types of DBMS helps in evaluating which database approach is appropriate for a particular requirement.

A DBMS, however, does not function through software alone. It operates within an environment involving physical resources, software, data, procedures, and different categories of users. These components and users are examined in the next section.

SGOU

1.1.3 Components and Users of DBMS

A DBMS does not operate in isolation. Managing a database requires a combination of physical resources, software, data, procedures, and people. Each component has a specific role, and the effective functioning of a database system depends on how these components work together.

Consider an online banking system. The database may contain millions of customer and transaction records, but the database system also requires servers and storage devices, DBMS software, application programs, security procedures, database administrators, programmers, and customers. Removing any major component can affect the functioning of the entire system.

Understanding these components provides a broader view of what constitutes a database environment.

1.1.3.1 Components of DBMS

The major components of a database environment are hardware, software, data, procedures, and people.

Hardware

Hardware refers to the physical devices used to operate and support the database system.

A database environment may require:

- servers
- storage devices
- client computers
- network equipment
- backup devices

Servers provide the processing environment in which the DBMS operates, while storage devices retain database files and related information.

For a large e-commerce platform, the database may need powerful servers and high-capacity storage because thousands of customers may access product and order information simultaneously.

The hardware requirements depend on factors such as database size, number of users, workload, performance expectations, and availability needs.

Software

Software provides the instructions and programs required to operate the database environment.

SGOU

The most important software component is the DBMS itself. However, a complete database environment may also involve:

- operating system software
- application programs
- database utilities
- network software
- backup and recovery software

The DBMS manages the database and provides facilities for operations such as data retrieval, data modification, security management, transaction processing, and recovery.

Application programs provide user-oriented functionality. For example, an online shopping application may provide facilities for searching products, placing orders, making payments, and tracking deliveries. The application communicates with the database through the DBMS.

Data

Data is acceptable in computing contexts.

The data stored in a database depends on the application. An airline reservation system may store information about flights, passengers, seats, and bookings. A hospital system may store information about patients, doctors, appointments, prescriptions, and medical reports.

In addition to the actual data, a database system maintains metadata.

Metadata means data about data. It describes characteristics of the data stored in the database.

For example, information about the name of a table, the names of its columns, their data types, and associated constraints constitute metadata. The DBMS uses such information to understand and manage the database structure.

Procedures

Procedures are documented instructions, rules, and guidelines for operating and maintaining the database system.

Procedures may specify how users are granted access, how backups are performed, how data is recovered after failures, how database changes are authorised, and how security incidents are handled.

The importance of procedures becomes clear when considering a critical application such as banking. Even if the DBMS and hardware are reliable, poorly defined procedures for backup, access control, or recovery can place the database at risk.

SGOU

Therefore, database management requires not only technology but also clearly defined operational practices.

People

People are an essential component of a database environment. Different individuals interact with the system according to their responsibilities and expertise.

Some users design the database, some develop applications, some manage the database environment, and others simply use the applications to perform their daily activities.

These different roles are discussed in detail below.

1.1.3.2 Database Users and their Roles

A database may be used by people with very different levels of technical knowledge. A customer using a banking application does not need to understand database structures, while a database administrator requires extensive technical knowledge.

Database users can therefore be classified according to their roles and the way they interact with the database.

Database Administrator

A Database Administrator (DBA) is responsible for the overall management and control of a database environment.

The responsibilities of a DBA may include:

- managing user accounts and privileges
- implementing database security
- monitoring database performance
- managing backup and recovery
- managing database storage resources
- maintaining database availability
- monitoring database operations
- ensuring appropriate database maintenance

Consider a hospital database containing sensitive patient information. The DBA may be responsible for ensuring that doctors, nurses, billing staff, and administrative personnel receive only the access required for their respective responsibilities.

The DBA also plays an important role in maintaining the reliability and availability of the database.

SGOU

Database Designer

A Database Designer is responsible for determining how the data should be organised within the database.

The designer studies the requirements of the organisation and identifies:

- entities
- attributes
- relationships
- constraints
- appropriate database schemas and structures.

For example, while designing an online shopping database, the designer may identify Customer, Product, Order, Payment, and Delivery as important entities and determine how they are related.

Good database design reduces unnecessary redundancy, supports data integrity, and makes data retrieval and maintenance more effective.

Application Programmer

An application programmer develops programs that interact with the database.

Most users do not directly interact with the DBMS. Instead, they use application programs that provide suitable interfaces.

Consider a railway reservation application. A programmer may develop the functionality required for searching trains, checking seat availability, making reservations, cancelling tickets, and displaying booking information. These operations require the application to communicate with the database.

Application programmers therefore need to understand how applications interact with database systems and how database operations can be incorporated into software.

End Users

End users are people who use the database system to perform their regular activities.

End users do not necessarily need detailed knowledge of database implementation. They often interact with databases through application interfaces.

For example, a customer using an online shopping application can search for products, place orders, view previous purchases, and track deliveries. The customer is an end user of the database system even though they may never directly write a database query.

End users can have different patterns of database usage.

SGOU

Naïve (parametric) users repeatedly perform predefined operations. A bank employee using a standard application screen to retrieve customer account information is an example.

Casual users access the database occasionally and may perform different queries depending on their requirements. A manager who retrieves sales information to prepare a monthly report can be considered a casual user.

Specialised Users

Specialised users work with database systems for specialised applications that may involve complex data or advanced processing requirements.

Examples include users working with:

- scientific databases;
- geographic information systems;
- multimedia databases;
- computer-aided design systems;
- artificial intelligence applications.

Such users may require database capabilities beyond those needed by ordinary business applications.

1.1.4 Data and Database Design

A database is useful only when the data stored in it is organised in a way that supports the requirements of the application. However, not all data has the same structure. Some data follows a clearly defined format, while other data may have a flexible or completely unstructured form.

Consider an online shopping platform. Customer names, order numbers, product prices, and order dates can be stored in clearly defined fields. At the same time, product reviews may contain text of different lengths, photographs may have different formats, and promotional videos may contain completely different types of content. Treating all these forms of data in exactly the same way would not be appropriate.

Understanding the nature of data is therefore important before designing a database. Database design determines how the required data will be organised, related, and maintained so that the resulting database can support the intended application effectively.

1.1.4.1 Structured, Semi-Structured and Unstructured Data

Data can broadly be classified as structured, semi-structured, and unstructured according to the extent to which it follows a predefined organisation.

SGOU

Structured Data

Structured data follows a predefined and well-defined format. The data is organised according to specific fields, data types, and relationships.

A relational database is a common example of an environment used to manage structured data. A customer record may contain fields such as Customer_ID, Name, Email, Phone, and Address. Each field has a defined purpose and generally follows a specified data type.

For example:

Customer_ID	Name	Email	Phone
C101	Anu	anu@example.com	9876543210
C102	Ravi	ravi@example.com	9876543211

The structure of the records is clearly defined. Each row follows the same general arrangement of attributes.

Structured data is relatively easy to search, sort, analyse, and process because its organisation is known in advance.

Examples include:

- bank account records
- employee records
- product catalogues
- examination marks
- sales transactions
- inventory records

Structured data is particularly suitable for applications where information can be represented using clearly defined attributes and relationships.

Semi-Structured Data

Semi-structured data does not follow a rigid tabular structure, but it contains certain organisational elements such as tags, keys, attributes, or other markers that provide information about the data.

Semi-structured data is useful when records may contain different attributes or when the structure needs to be more flexible.

For example, two products in an online shopping system may not have exactly the same characteristics. A laptop may have RAM and processor details, while a shirt may have size and fabric information. A flexible representation can accommodate these differences more easily.

SGOU

Examples of semi-structured data include:

- XML documents;
- JSON documents;
- email messages;
- configuration files;
- data exchanged between web applications.

Unstructured Data

Unstructured data does not follow a predefined data model or a fixed organisational structure suitable for traditional tabular storage.

Examples include photographs, audio recordings, videos, social media posts, scanned documents, presentations, and text documents.

Consider a hospital. Patient information may include structured data such as patient ID, age, appointment date, and billing amount. However, medical images, scanned reports, and recorded consultations may not fit naturally into the same tabular structure.

Unstructured data is increasingly important because organisations generate large quantities of multimedia and textual information. Managing such data often requires specialised technologies and storage approaches.

Comparison of Data Types

The three forms of data differ mainly in the degree of structure they possess.

Aspect	Structured Data	Semi-Structured Data	Unstructured Data
Organisation	Fixed and predefined	Flexible but contains organisational markers	No predefined structure
Common representation	Tables	JSON, XML	Text, image, audio, and video
Schema	Usually defined in advance	Flexible or partially defined	No fixed schema for the content
Processing	Relatively straightforward	Requires flexible processing	Often requires specialised processing
Example	Bank transaction	JSON product record	Product image

SGOU

The distinction between these categories is not always absolute. A single application may contain all three types of data. For example, an e-commerce system may store customer and order information as structured data, product details as semi-structured data, and product images and customer reviews as unstructured or semi-structured content depending on format.

Therefore, understanding the nature of the data is an important consideration when selecting technologies and designing a database.

Once the required data has been identified and its characteristics understood, the next question is: How should that data be organised within the database? This is the purpose of database design.

1.1.4.2 Database Design

Database design is the process of determining how data should be organised, related, and constrained to meet the requirements of an application.

A well-designed database should represent the important aspects of the real-world system accurately while reducing unnecessary redundancy and maintaining data integrity.

Consider an online shopping application. Before creating the database, it is necessary to determine what information needs to be stored. The system may need information about customers, products, orders, payments, and deliveries.

Simply creating tables without first understanding these requirements can result in an inefficient or incorrect database. Database design therefore begins with understanding the application and identifying the data it needs.

A good database design generally aims to:

- represent the required data accurately;
- establish meaningful relationships among data;
- minimise unnecessary redundancy;
- maintain data integrity;
- support efficient data retrieval;
- provide appropriate security;
- accommodate future requirements where possible.

Database design can be understood through three broad stages/phases: conceptual design, logical design, and physical design.

SGOU

1.1.4.3 Conceptual, Logical and Physical Database Design

Conceptual Database Design

Conceptual database design provides a high-level representation of the data requirements of an organisation or application.

At this stage, attention is focused on understanding the important entities, their attributes, and the relationships among them rather than on the technical details of storage.

For an online shopping system, important entities might include Customer, Product, Order, Payment, and Delivery.

A customer may place one or more orders. An order may contain one or more products. A payment may be associated with an order, and delivery information may be associated with the order.

An Entity-Relationship (ER) model is commonly used during conceptual database design to represent entities, attributes, and relationships.

The conceptual design therefore answers an important question:What information does the system need to represent, and how are the different data elements related?

Logical Database Design

Logical database design converts the conceptual representation into a structure that can be implemented using a particular type of database model.

For a relational database, entities identified during conceptual design are generally represented as tables, and their attributes become columns.

For example, the Customer entity may be represented using attributes such as Customer_ ID, Name, Email, and Phone. The Order entity may contain Order_ID, Order_Date, Customer_ID, and other relevant attributes.

Keys are also identified at this stage. A **primary key** uniquely identifies a record within a table. A **foreign key establishes** a relationship referencing the primary key (or candidate key) of another table.

The logical design must also consider constraints and relationships so that invalid or inconsistent data can be prevented.

Physical Database Design

Physical database design deals with how the logically designed database will actually be stored and accessed within a particular database environment.

It considers issues such as:

- storage structures
- indexes

SGOU

- file organisation
- partitioning
- access paths
- storage allocation
- performance requirements

For example, if an online shopping system contains millions of orders, appropriate indexing can help retrieve orders for a particular customer more efficiently.

Physical design is therefore concerned with implementation and performance rather than simply identifying what data should exist.

The three stages can be viewed as different levels of the same design process. Conceptual design focuses on the requirements and meaning of the data. Logical design determines how those requirements can be represented using a database model. Physical design considers how the resulting database can be implemented efficiently within a particular database system.

A carefully designed database provides a strong foundation for data management. Poor design, on the other hand, can lead to redundancy, inconsistency, inefficient queries, and difficulties in maintaining the system.

Once the data has been appropriately organised and the database structure has been designed, users need a way to define, retrieve, modify, and control the data. This brings us to the role of database languages, which provide the means for performing these operations.

1.1.5 Database Languages

A database becomes useful when users and applications can interact with the data stored in it. They may need to create database structures, add new records, retrieve information, modify existing records, or control access to the data. These operations require a systematic way of communicating with the DBMS.

Database languages provide this means of communication. They consist of commands used to define and manipulate database structures and data, retrieve information, control access, and manage transactions.

SQL (Structured Query Language) is the most widely used database language in relational database systems. Although SQL is commonly discussed as a single language, its commands can be grouped according to the type of operation they perform.

Understanding these categories makes it easier to identify the purpose of different database commands.

SGOU

1.1.5.1 Database Languages and their Purposes

Database languages can be classified according to the type of operation they perform on a database. In a relational database environment, SQL provides commands for defining database structures, manipulating stored data, retrieving information, controlling access, and managing transactions. Based on these purposes, SQL commands are commonly discussed under different categories.

Data Definition Language (DDL) is concerned with defining and modifying the structure of database objects. It is used when the structure of a database needs to be created or changed.

Data Manipulation Language (DML) is concerned with inserting, retrieving,modifying, and deleting data stored in database objects.

Data Control Language (DCL) deals with controlling access to database resources. It allows appropriate privileges to be granted to or removed from users.

Transaction Control Language (TCL) deals with managing transactions and controlling how changes made during database operations are committed or reversed.

The following table provides a brief overview of these categories. The individual commands, their syntax, and their practical applications will be studied in detail in Block 2, Unit 2.

Database LanguageCategory	Main Purpose
DDL	Defines and modifies the structure of database objects
DML	Inserts, modifies, and deletes stored data
DCL	Controls user access and privileges
TCL	Manages database transactions and their changes

Thus, database languages provide the means for interacting with a database at different levels. At this stage, it is sufficient to understand the purpose of each category and how they contribute to database management. The detailed study of SQL commands and their usage will be taken up later in the course.

SGOU

Summarised Overview

This section introduces the fundamental concepts of database management and explains why DBMSs are required for managing large and shared collections of data. It begins by distinguishing data, information, and databases and discusses the limitations of traditional file-based approaches. The major characteristics of a DBMS, including data organisation, sharing, security, integrity, data independence, concurrency control, backup, recovery, and consistency, are then explained.

The section also introduces different types of DBMS, including hierarchical, network, relational, object-oriented, object-relational, and NoSQL database systems. It further examines the components of a database environment and explains the roles of database administrators, database designers, application programmers, end users, and specialised users.

The different forms of data—structured, semi-structured, and unstructured - are introduced along with their characteristics and examples. The section then explains database design through its conceptual, logical, and physical stages. Finally, it introduces database languages and their major categories: DDL, DML, DCL, and TCL, while noting that their detailed commands and practical usage will be studied later.

SGOU

Assignments

1. Explain the need for a Database Management System. Discuss the major problems associated with managing data using a traditional file-based approach.

2. Describe the major characteristics of a DBMS. Explain how data security, data integrity, data independence, concurrency control, and backup and recovery contribute to effective database management.

3. Explain the three stages of database design with suitable examples.

4. An online shopping system stores customer, product, order, payment, and delivery information. Analyse how a poorly designed database could lead to redundancy, inconsistency, security problems, and inefficient data retrieval. Suggest how appropriate database design can address these issues.

5. A social networking application needs to manage millions of users and their complex relationships, while a banking application needs to manage highly structured financial transactions. Analyse the suitability of different types of DBMS for these applications and justify your selection based on their requirements.

Reference

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database system concepts* (7th ed.). McGraw-Hill.

2. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2009). *Database systems: The complete book* (2nd ed.). Pearson.

3. Date, C. J. (2004). *An introduction to database systems* (8th ed.). Pearson.

4. Oppel, A. J. (2018). *Databases demystified* (2nd ed.). McGraw-Hill Education.

Suggested Reading

1. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage.

2. Connolly, T., & Begg, C. (2015). *Database systems: A practical approach to design, implementation, and management* (6th ed.). Pearson.

3. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill.

4. Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

2 UNIT

Data Models and Schemas

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the concept and types of Data Models and database schemas
- classify database system architectures based on their organisation and functionality
- differentiate the levels of database abstraction and their purposes
- analyse the role of data independence in database system design
- select suitable data models and architectural approaches for different database requirements

SGOU

Background

Modern applications generate and use large volumes of data in different forms. An online shopping application manages information about customers, products, orders, and payments, while a social networking platform maintains user profiles and complex relationships among users. Before such information can be stored and managed effectively, it is necessary to determine how the data should be represented, organised, related, and accessed.

A data model provides the concepts required to describe the structure of data and the relationships among different data elements. Once a suitable model is selected, the database structure can be formally defined through a schema, while the actual data stored in that structure forms the database instance. The organisation of thedatabase environment is determined through appropriate database system architectures.

Database systems also manage complexity by presenting data at different levels of abstraction. This separation makes it possible to hide unnecessary implementation details and supports data independence, allowing certain changes to be made without significantly affecting applications or users.

Understanding these concepts provides an essential foundation for database design and helps learners examine how data requirements can be translated into an organised and adaptable database system.

Keywords

Data Model, Database Schema, Database Instance, Database Architecture, Data Abstraction, Data Independence, Physical Data Independence, Logical Data Independence

Discussion

SGOU

1.2.1 Data Models

When a database is designed, it is not enough to decide what data needs to be stored. It is also necessary to determine how that data should be represented and how different data items are related to one another. A data model provides a systematic way of describing these aspects.

Consider an online food delivery application. The system needs to maintain information about customers, restaurants, menu items, orders, payments, and deliveries. A customer can place multiple orders, an order can contain several menu items, and each order may have a corresponding payment and delivery. Before such information is stored in a database, these entities and their relationships need to be identified and represented in a suitable manner.

A data model is a collection of concepts used to describe the structure of a database, the relationships among data, the constraints on the data, and, depending on the model, the operations that can be performed on the database.

Data models provide an organised way to represent a real-world system in a form that can be understood and implemented within a database environment. They act as a bridge between the requirements of an application and the actual database structure.

1.2.1.1 Concept and Purpose of Data Models

A data model describes how data is organised and how different data elements are connected. It provides concepts for identifying the important objects or entities in a system, describing their properties, and representing relationships among them.

For example, in a railway reservation system, important entities may include Passenger, Train, Station, and Reservation. A passenger may make a reservation for a particular train, and a train may operate between several stations. A data model helps represent these entities and the relationships among them in a structured manner.

A data model generally addresses three important aspects of a database:

Data structure describes how data is organised. Depending on the model, data may be represented as tables, records, objects, documents, graphs, or other structures.

Relationships describe how different data items are associated. For example, a customer may place several orders, while each order may contain several products.

Constraints specify rules that the data should satisfy. For example, a customer identifier may be required to be unique, or an order may be required to belong to an existing customer.

Some data models also provide concepts for describing the operations that can be performed on the represented data.

The primary purpose of a data model is therefore to provide a clear and systematic description of data before and during database development. It helps database designers understand the requirements of an application and translate those requirements into an appropriate database structure.

A suitable data model can also improve the quality of database design. When entities, relationships, and constraints are clearly identified, problems such as unnecessary duplication, inappropriate relationships, and invalid data can be identified at an early stage.

Data models are also useful for communication. Database designers, application developers, analysts, and other stakeholders may have different technical backgrounds. A well-defined model provides a common representation through which the structure and requirements of the database can be discussed.

It is important to understand that a data model is not the same as the actual database. A data model provides the concepts and rules for describing data, whereas a database contains the actual data organised according to those concepts.

For example, the relational model defines concepts such as tables, rows, columns, keys, and relationships. A particular banking database can then be created using those concepts to store actual customer accounts and transaction records.

SGOU

Data models can be considered at different levels of detail. A high-level model focuses on the important entities and relationships in the real-world system, while a more implementation-oriented (low-level)model describes how the data can be represented within a particular database system.

This leads to the classification of data models into different types.

1.2.1.2 Types of Data Models

A data model provides a way of representing the data requirements of a real-world system. Different approaches have been developed for representing data, depending on the nature of the information and the relationships that need to be expressed.

The earlier discussion introduced different types of DBMS based on the approaches they use. Here, the focus is different. The purpose is to understand how data itself can be represented using different modelling approaches and how these approaches help in designing a database.

A data modelling approach determines the basic concepts used to describe data and the relationships among data items. For example, one approach may represent information using tables, while another may use a hierarchy, interconnected records, objects, documents, or graphs.

1. Hierarchical Approach / Hierarchical Model

The hierarchical approach represents data in a tree-like arrangement. Data items are organised through parent-child relationships, creating different levels within the structure.

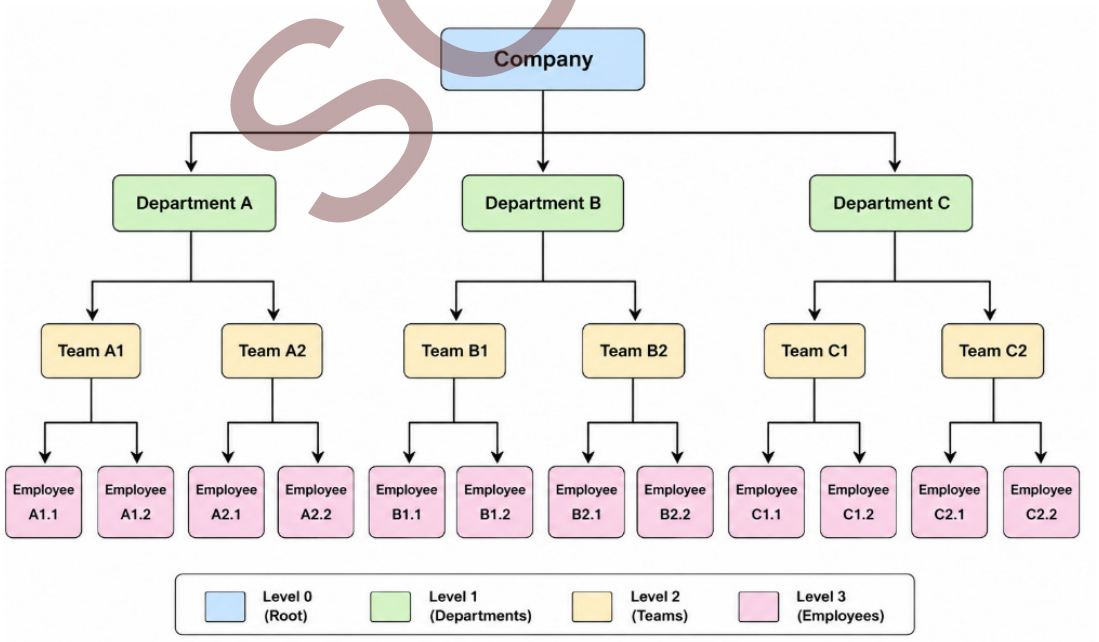


Fig.1.2.1 Hierarchical Model

SGOU

Consider a company's organisational information. A company may have several departments, and each department may have several employees. The relationship between the company, departments, and employees can be represented as a hierarchy.

The hierarchical approach is appropriate when relationships naturally follow a parent-child structure. However, it becomes less convenient when a data item needs to have multiple independent relationships with other data items.

2. Network Model

The networkapproach represents data as interconnected records and allows a data item to participate in multiple relationships.

Consider a project management application. An employee can work on several projects, and each project can involve several employees. The network approach can represent these many-to-many relationships more naturally than a strict hierarchy.

This approach provides greater flexibility in representing complex connections among data items.

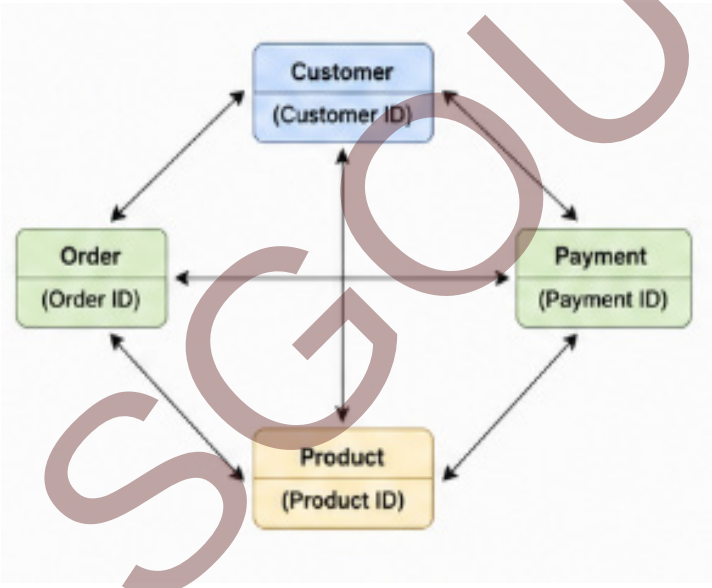


Fig.1.2.2 Network Model

3. Relational Model

The relational approach represents data using relations, which are commonly implemented as tables. Each table contains attributes represented by columns and individual records represented by rows.

Consider a product database:

Product_ID	Product_ Name	Category	Price
P101	Laptop	Computer	65000
P102	Keyboard	Accessory	2500
P103	Monitor	Computer	12000

SGOU

Here, Product_ID, Product_Name, Category, and Price represent attributes, while each row represents a particular product.

Real-world information is generally distributed across multiple related tables rather than being stored in one large table. For example, customer information can be stored separately from order information, with appropriate relationships established between them.

The relational approach is particularly suitable for structured data and applications where relationships and data integrity need to be clearly defined.

Concepts such as relations, attributes, keys, integrity constraints, and relational operations form an important part of relational data modelling and will be explored further in later units.

4. Entity-Relationship Model

The Entity-Relationship (ER)approach provides a conceptual way of representing the data requirements of a system.

It focuses on identifying entities**,** attributes**,** and relationships.

Consider an online food delivery system. Customer, Restaurant, Menu Item, Order, and Delivery can be identified as important entities. A customer places an order, an order contains menu items, and an order is associated with a delivery. These relationships can be represented using an ER model.

The ER approach is particularly useful during the early stages of database design because it allows the requirements of a system to be represented at a high level before implementation details are considered.

The ER approach therefore helps answer questions such as:

- What are the important objects in the system?
- What information needs to be maintained about them?
- How are these objects related?
- What rules or constraints apply to these relationships?

5. Object-Oriented Model

The object-oriented approach represents data as objects and uses concepts such as classes, inheritance, and encapsulation.

An object can contain both its data and the characteristics or operations associated with that data. This approach can be useful when the application itself is developed using object-oriented programming and needs to handle complex data structures.

For example, a computer-aided design application may contain objects representing buildings, rooms, doors, windows, and other design elements. These objects may have

SGOU

different properties and relationships and may inherit common characteristics from other objects.

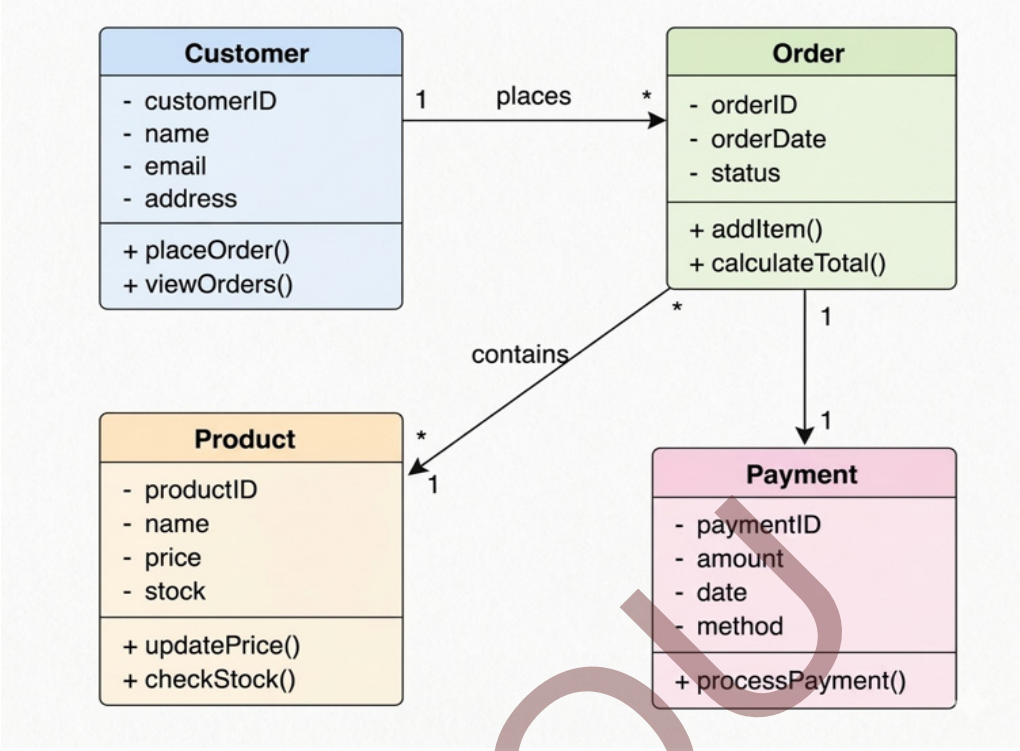


Fig.1.2.3 Object-oriented model

The object-oriented approach is therefore particularly useful for applications where data has complex structures that are difficult to represent using simple records and tables.

6. Graph Model / Graph Approach

The graph approach represents data in terms of nodes and relationships, commonly called edges.

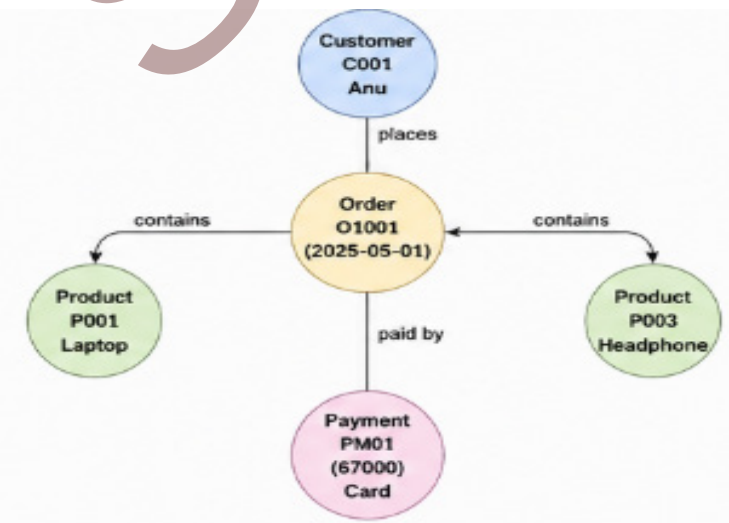


Fig.1.2.4 Graph model

SGOU

Consider a social networking application. Users can be represented as nodes, while relationships such as friendship, following, or professional connections can be represented as edges.

This approach is particularly useful when relationships among data items are as important as the data items themselves. Recommendation systems, social networks, fraud detection, and network analysis are examples of areas where graph-oriented representations can be valuable.

1.2.2 Database Schema and Instance

A data model provides the concepts for representing data and the relationships among data items. Once these concepts are used to describe a particular database, it becomes necessary to distinguish between the structure of the database and the actual data stored in that structure.

Consider an online food delivery system. The database may be designed to maintain information about customers, restaurants, orders, payments, and deliveries. The definitions of these data elements, their attributes, relationships, and constraints form the structure of the database. At a particular moment, the database may contain thousands of actual customer and order records. These records can change continuously as new customers register, orders are placed, payments are made, and deliveries are completed.

The structure remains relatively stable, while the actual contents change frequently. This distinction is expressed through the concepts of database schema and database instance.

1.2.2.1 Database Schema

A database schema is the formal description of the structure of a database, including its tables, attributes, relationships, and constraints. It describes how data is organised and how different components of the database are related.

A schema may specify:

- the tables or other database structures
- attributes and their data types
- relationships among data items
- keys
- integrity constraints
- other structural elements supported by the database system

For example, an online food delivery database may have tables such as Customer, Restaurant, Order, Payment, and Delivery. The schema defines the attributes associated with these tables and the relationships among them.

SGOU

The schema can therefore be viewed as a blueprint of the database. It describes what kind of data can be stored and how that data is organised, rather than containing the actual values themselves.

Suppose a Customer table is defined with the following attributes:

Customer_ID	Name	Phone	Email
—	—	—	—

The definition of these attributes, their data types, and constraints forms part of the database schema. The actual customer values stored in the table are not the schema itself.

A schema generally changes less frequently than the data stored in the database. For example, thousands of new orders may be added to an e-commerce database without changing the basic structure of the Order table.

The schema is therefore important for maintaining a consistent organisation of data and for ensuring that users and applications interact with the database according to a defined structure.

In a DBMS, schemas are often described at three levels: external schema, conceptual schema, and internal schema.

1.2.2.2 Database Instance

A database instance refers to the actual collection of data stored in the database at a particular point in time.

Unlike a schema, an instance changes frequently as database operations are performed.

Consider the Customer table defined earlier. At one particular moment, it may contain:

Customer_ ID	Name	Phone	Email
C101	Anu	9876543210	anu@example.com
C102	Ravi	9876543211	ravi@example.com

These actual records constitute part of the database instance.

If another customer registers, a new record is added and the instance changes. If a customer updates the customer's phone number, the instance changes again. If a customer record is deleted, the instance is modified once more.

The schema, however, may remain unchanged throughout these operations.

This distinction can be understood through a simple observation: the schema describes the structure, while the instance represents the data occupying that structure at a particular time.

SGOU

A database may therefore have the same schema while having many different instances over its lifetime.

For example, an online shopping database may have a fixed structure containing Customer, Product, Order, and Payment tables. The actual records in these tables on Monday will differ from those on Friday because new customers, products, and orders may have been added and existing information may have been modified.

Having understood the distinction between schema and instance, the next step is to examine how a database system is organised to serve users and applications. This leads to database system architectures and their classification.

1.2.3 Database System Architectures and Classification

A database does not operate in isolation. It forms part of a larger computing environment in which users, application programs, database software, and storage systems work together. The way these components are organised and communicate with one another is referred to as database system architecture.

Database architecture determines where the database is located, where processing takes place, how users access the database, and how different software components interact. The choice of architecture can affect system performance, scalability, security, maintenance, and accessibility.

For example, consider an online university registration system. Students may use a web or mobile application to register for courses, while the application communicates with a database containing student, course, registration, and examination information. The database may be located on the same computer in a small system or on dedicated servers in a large university environment. These different arrangements represent different database architectures.

Database system architectures can be classified based on how data processing and communication are distributed among users, application programs, and database servers.

1.2.3.1 Database System Architecture

Database system architecture describes the overall organisation of the components that participate in database processing. The major components generally include:

1. users who interact with the system
2. application programs that provide services to users
3. the DBMS that manages database operations
4. the database containing stored data and

SGOU

5. supporting hardware and network infrastructure

The architecture defines how these components are distributed and how requests move between them.

A simple database application may have the user interface, application logic, DBMS, and database located on a single computer. In a larger system, these components may be distributed across multiple computers or servers.

Database architectures can therefore be understood from different perspectives. One perspective considers where the components are located, while another considers how application processing is divided among different layers.

The following sections examine commonly used architectural arrangements.

Based on these perspectives, database system architectures are commonly classified as centralised, client-server, two-tier, and three-tier architectures.

1.2.3.2 Centralised and Client–Server Architectures

Centralised and client–server architectures represent two important ways of organising database processing.

1. Centralised Database Architecture

In a centralised database architecture, the database and the major database processing functions are located at a central computer or server. Users access the central system through terminals or connected applications.

For example, a small organisation may maintain its employee database on a single server. Employees use applications connected to this server to retrieve or update information.

The centralised approach provides a relatively simple environment for administration because data and database management are concentrated in one location. Backup, security, and maintenance can also be managed centrally.

However, heavy dependence on the central system can become a limitation. If the central server becomes unavailable, users may lose access to database services. Performance may also become a concern when a large number of users access the system simultaneously.

2. Client–Server Database Architecture

In a client–server architecture, database processing is distributed between client systems and a database server.

The client is generally responsible for interacting with the user and sending requests to the server. The database server runs the DBMS, processes database requests, and manages access to stored data.

For example, in a college examination system, an application running on an administrator's computer may request student examination information from a database

SGOU

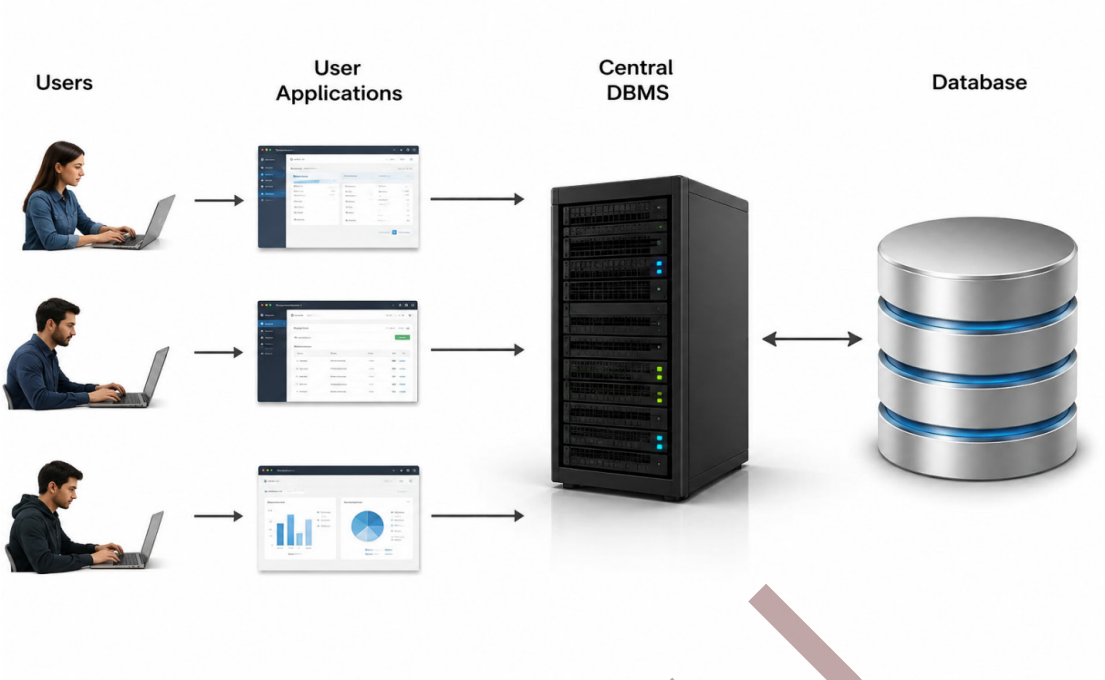


Fig. 1.2.5 Centralised Database Architecture

server. The server processes the request and returns the required result.

This architecture allows several clients to access a common database while centralising database management on the server. It is widely used in organisational applications where many users need shared access to data.

The distinction between the two approaches can be summarised as follows:

Aspect	Centralised Architecture	Client–Server Architecture
Main processing location	Central system	Shared between client and server
Database location	Central system	Database server
User access	Through connected terminals/applications	Through client applications
Administration	Highly centralised	Mainly managed at the server side
Typical use	Smaller or highly centralised systems	Multi-user organisational systems

As database applications became more complex, simply dividing processing between a client and a server was often insufficient. Applications began to separate user-interface functions, application processing, and database management into distinct layers. This led to the development of multi-tier architectures.

SGOU

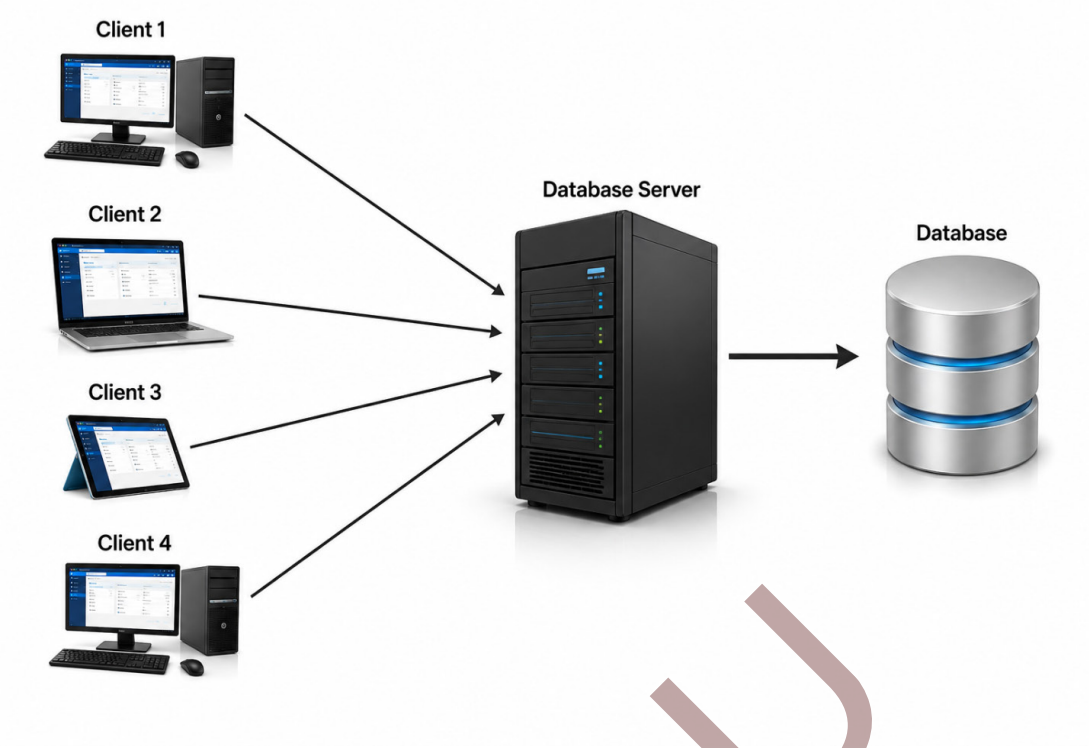


Fig.1.2.6 Client-Server Architecture

1.2.3.3 Two-Tier and Three-Tier Architectures

Two-tier and Three-tier architectures describe how an application is divided into logical processing layers.

Two-Tier Architecture

In a two-tier architecture, the application is divided into two main layers:

1. Client layer – provides the user interface and usually contains some application logic.

2. Database server layer – manages the DBMS and database.

The client communicates directly with the database server.

For example, a desktop-based library application may allow a librarian to search for books and issue or return them. The application running on the librarian's computer communicates directly with the database server.

Two-tier architecture is relatively simple to develop and can provide efficient communication in smaller environments. However, when the number of clients becomes very large, maintaining application logic on individual client systems can become difficult.

SGOU

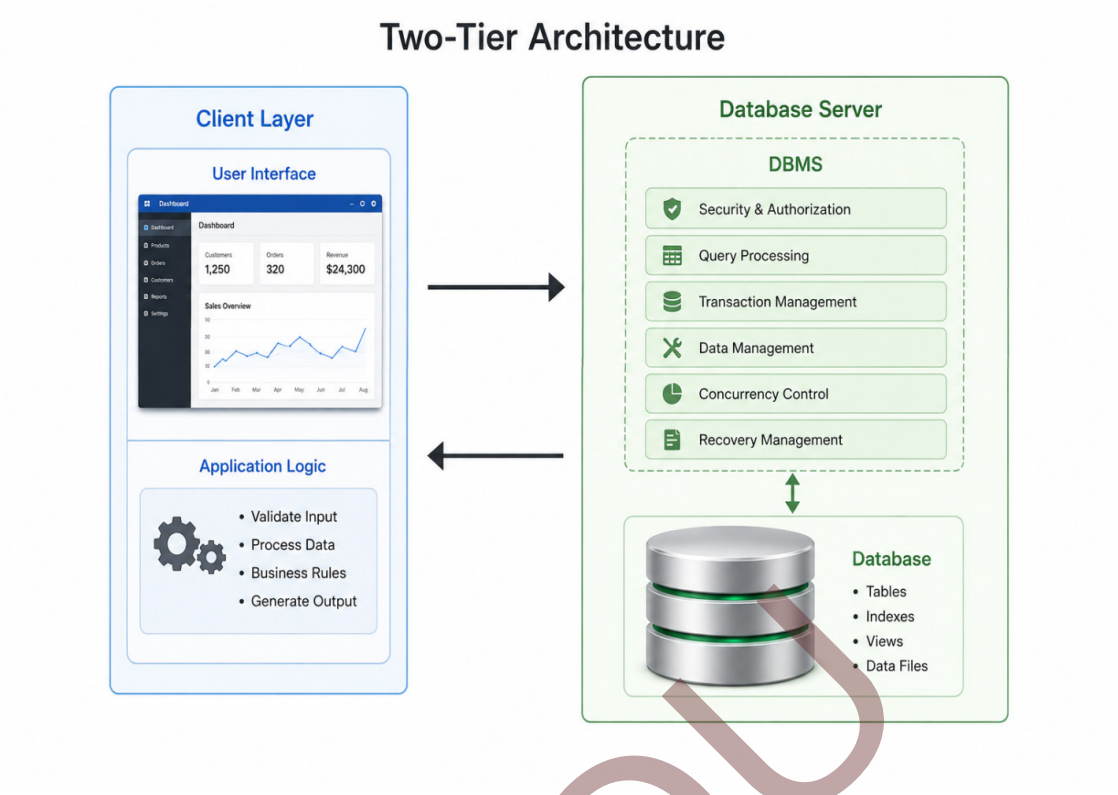


Fig.1.2.7 Two-Tier Architecture

Three-Tier Architecture

A three-tier architecture introduces an intermediate application layer between the client and the database server. The three layers are:

1. Presentation layer – provides the user interface.
2. Application or business logic layer – processes requests and applies application rules.
3. Database layer – manages data through the DBMS.

The arrangement can be represented as:

Consider an online learning platform. A learner uses a web browser to view course information. The request is sent to the application server, which checks the learner's permissions, processes the request, and communicates with the database to retrieve the required information. The result is then returned through the application layer to the learner.

Three-tier architecture provides several advantages. Application logic can be managed centrally, database details can be separated from the user interface, and the system can be scaled by adding or upgrading application servers. It can also support stronger security because clients do not need direct access to the database.

The choice between two-tier and three-tier architecture depends on the size, complexity, number of users, security requirements, and expected growth of the application.

SGOU

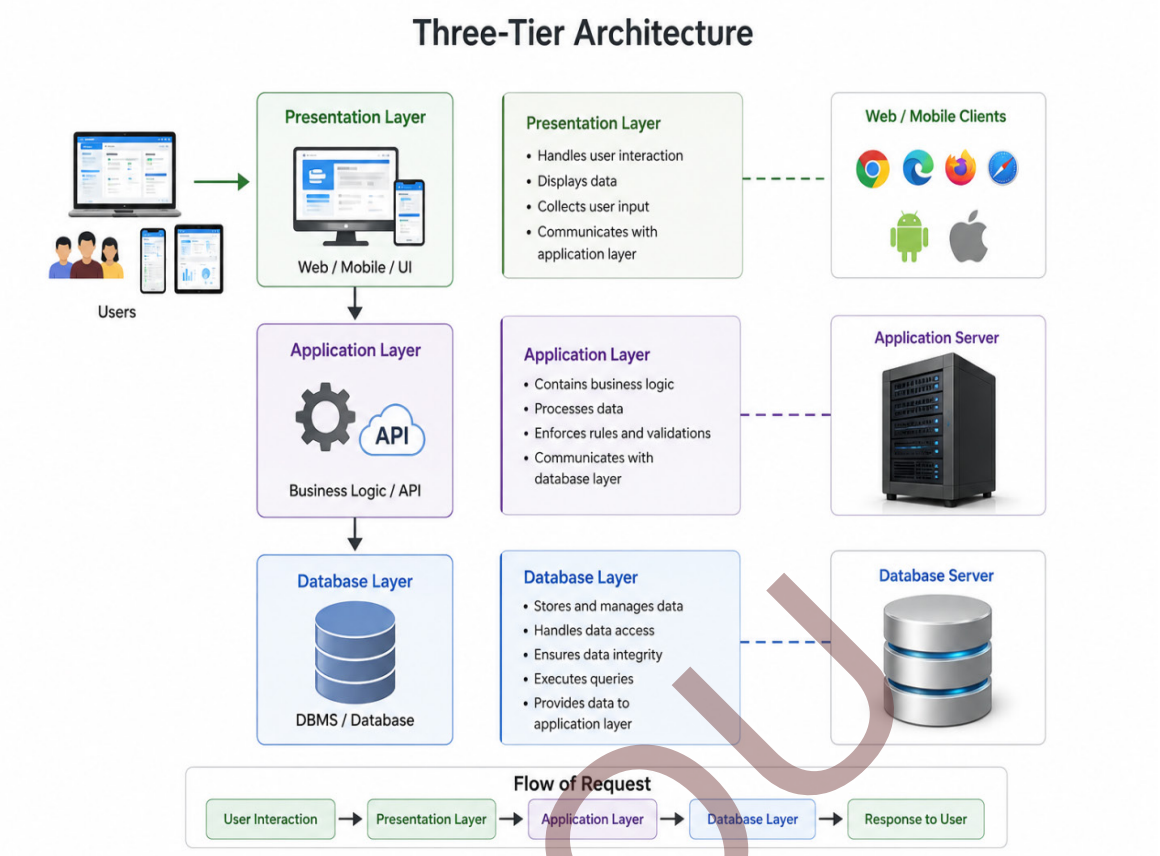


Fig.1.2.8 Three-Tier Architecture

Aspect	Two-Tier	Three-Tier
Layers	2	3
Client access	Direct database access	Through application server
Security	Lower	Higher
Scalability	Limited	Better
Common use	Small applications	Enterprise applications

1.2.3.4 Classification of Database Systems

Database systems can be classified using different criteria. No single classification is sufficient for every situation because a database system can be described according to its location, processing arrangement, number of users, or the way data is distributed.

Based on the location and distribution of data, database systems can be broadly classified as follows:

Centralised Database System

A centralized database system stores and manages the database at a single location. Users may access the database from different terminals or applications, but the primary database is maintained centrally.

SGOU

This arrangement simplifies administration and provides centralised control over data. It is suitable when an organisation prefers to maintain its database in one controlled environment.

Distributed Database System

A distributed database system stores data across multiple network-connected locations while allowing the collection of databases to function as a coordinated system.

For example, a bank may maintain databases at different regional locations while allowing authorised users to access required customer information across branches.

A distributed system can improve availability and support geographically distributed operations. However, maintaining consistency, coordinating transactions, securing communication, and managing failures across locations can be more complex.

Parallel Database System

A parallel database system uses multiple processors or computing resources to perform database operations simultaneously. Large queries or data-processing tasks can be divided into smaller operations that are executed in parallel.

This approach is useful when very large volumes of data need to be processed efficiently. Data analytics, scientific applications, and large-scale information systems can benefit from parallel database processing.

Cloud Database System

A cloud database system provides database services through cloud computing infrastructure. The database may be hosted and managed using cloud resources rather than being maintained entirely on an organisation's own physical infrastructure.

Cloud databases can provide flexible resource allocation, scalability, remote accessibility, and managed database services. They are commonly used for web applications, enterprise systems, analytics platforms, and other applications requiring flexible computing resources.

Type	Main Idea	Example
Centralised	Database at one location	Small organisation database
Distributed	Data stored at multiple locations	Banking branches
Parallel	Multiple processors execute tasks	Data analytics
Cloud	Database hosted on cloud infrastructure	Web applications

1.2.4 Database Abstraction Levels

Database systems can contain a large amount of information about how data is organised and stored. However, every user of a database does not need to know all these details. A student using a university portal, for example, is interested in viewing

SGOU

course registrations and examination results, but does not need to know where the corresponding records are physically stored or how disk blocks are organised.

Database abstraction addresses this issue by hiding unnecessary implementation details and presenting the database at different levels of detail. It allows different users to work with the database according to their requirements while the underlying storage mechanisms remain hidden. It reduces the complexity of database interaction by separating what users need to know from how the data is actually managed.

A database system therefore provides different levels of abstraction. At a higher level, users work with relevant information without being exposed to implementation details. At lower levels, database designers and administrators can work with the logical organisation and physical storage of the data.

The main objectives of data abstraction are to:

- hide unnecessary implementation details
- simplify interaction with the database
- reduce the complexity of database design and use
- provide appropriate views of the same data
- support controlled access to information and
- allow changes in storage and implementation without unnecessarily affecting users

Database systems commonly describe data using three levels of abstraction:

1. Physical Level
2. Logical Level
3. View Level

SGOU

1.2.4.2 Physical, Logical and View Levels

Database abstraction is organised into three levels: the physical level, the logical level, and the view level. Each level describes the database from a different perspective and hides details that are not relevant at that level.

Physical Level

The physical level, also known as the internal level, is the lowest level of database abstraction. It describes how data is physically stored and accessed by the database system.

This level deals with implementation details such as:

- organisation of database files;

- storage of records and data blocks;
- indexes and access paths;
- file organisation techniques; and
- storage and retrieval mechanisms.

These details are important for achieving efficient storage and fast data access. However, they are normally hidden from users and application programs.

For example, when a DBMS retrieves a particular record, several internal operations may be required to locate that record efficiently. The physical level deals with such operations without exposing them to the user.

Logical Level

The logical level, also called the conceptual level, describes the overall logical structure of the database. It identifies the data maintained by the system and the relationships and constraints associated with that data.

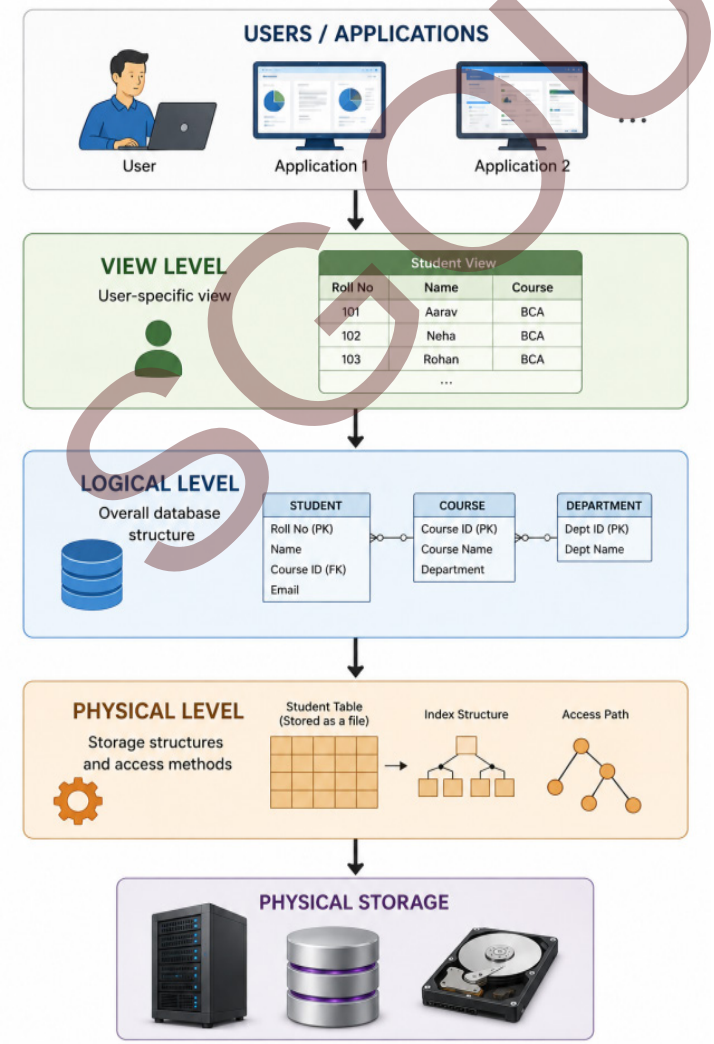


Fig. 1.2.9 Levels of Abstraction

SGOU

At this level, the database may be described in terms of:

- Entities (conceptual level) or tables (relational implementation);
- attributes;
- relationships;
- keys;
- integrity constraints; and
- other logical structures.

The logical level does not specify how these structures are physically stored. Its purpose is to provide a complete logical description of the database that can be used by database designers and application developers.

For instance, a database may contain a Student structure with attributes such as Student_ID, Name, Programme, and Email, together with relationships connecting students to courses and registrations. The logical level describes these structures and relationships without specifying their physical storage locations.

View Level

The view level, also called the external level, is the highest level of abstraction. It describes the part of the database that is relevant to a particular user or application.

A database may contain a large amount of information, but a particular user may require access to only a specific portion of it. The view level allows such requirements to be represented separately.

Different users can therefore have different views of the same database. A view may also combine information from multiple underlying structures and present it in a form that is convenient for a particular task.

The three levels can be compared as follows:

Level	Also Known As	Main Concern	Typical Focus
Physical	Internal level	How data is stored	Files, indexes, storage structures
Logical	Conceptual level	What data is stored and how it is related	Tables, attributes, relationships, constraints
View	External level	What information a user or application needs	User-specific representations

These levels are not three separate copies of the database. They are different ways of describing and accessing the same underlying database system.

In a university database, a student may see only personal details and examination results, while a faculty member may see course and student performance information. Both views are derived from the same underlying database.

SGOU

1.2.4.3 Mapping between Abstraction Levels

The three abstraction levels must be connected so that a request made through a user view can ultimately be processed against the physically stored data. The DBMS provides mappings between the levels to maintain this connection.

There are two important mappings:

1. View–logical mapping
2. Logical–physical mapping

View–Logical Mapping

The view–logical mapping connects an external view with the corresponding structures in the logical database.

A user view may contain only selected attributes or information required for a particular task. The DBMS determines how this information corresponds to the underlying logical structures.

For example, a result-reporting view may present a learner's course name, examination result, and grade even though these values are maintained through several logical structures. The mapping establishes the relationship between the user-facing representation and those logical structures.

Logical–Physical Mapping

The logical–physical mapping connects the logical database structure with its physical storage representation.

The logical level may define tables, attributes, relationships, and constraints, while the physical level determines how the corresponding records are organised and accessed in storage.

This mapping enables the DBMS to manage physical storage without requiring changes to the logical description whenever storage techniques are modified.It provides an important separation between the different levels. A change made at one level does not necessarily require corresponding changes at all other levels.

For example, a database administrator may change the way records are stored or introduce a more efficient indexing technique at the physical level. If the logical structure remains unchanged, applications using that logical structure can continue to operate without modification.

This separation between different levels forms the basis for data independence. Data independence means that changes at one level of the database can be made with minimal impact on the levels above it. The concept and its two major forms are discussed in the next section.

SGOU

1.2.5 Data Independence

Data independence is the ability to modify the structure of a database at one level of abstraction without requiring changes at the next higher level.

A database system may need to change over time because of increasing data volume, improved storage technology, changing application requirements, or performance considerations. If every such change required modifications to application programs and user views, database maintenance would become difficult and costly.

Data independence reduces this dependency by keeping the different levels of database abstraction sufficiently separate.

For example, a change in the physical organisation of database files should ideally not affect the logical structure used by applications. Similarly, certain changes to the logical database structure should not require every user view or application program to be redesigned.

Data independence is mainly classified into two types:

1. Physical data independence
2. Logical data independence

The distinction between them depends on the level at which the change occurs.

1.2.5.2 Physical Data Independence

Physical data independence is the ability to change the physical or internal level of a database without affecting the logical level.

Physical storage techniques may need to be modified to improve performance, increase storage efficiency, or make better use of available hardware. Such changes should not normally require changes to the logical database structure or application programs.

Examples of physical-level changes include:

- changing file organisation techniques
- creating or removing indexes
- changing storage allocation methods
- modifying access paths
- moving database files to different storage media.

Suppose an index is added to improve the speed of searching a large table. The physical storage and access mechanism changes, but the logical structure of the table remains the same. Applications can continue to use the table without knowing that a new indexing technique has been introduced.

SGOU

Physical data independence is generally easier to achieve because applications usually interact with the logical level rather than directly with physical storage structures.

1.2.5.3 Logical Data Independence

Logical data independence is the ability to change the logical or conceptual level of a database without requiring changes to external views or application programs that are not affected by the modification.

Logical changes can involve modifications to the structure of the database to accommodate new requirements.

Examples include:

- adding a new attribute to an existing structure
- introducing a new table or entity
- establishing a new relationship
- modifying certain logical constraints
- restructuring logical data while preserving the requirements of existing applications.

For example, suppose an existing academic database is extended to maintain an additional attribute for students. If existing applications do not depend on that attribute and their required views remain unchanged, those applications may continue to operate without modification.

Logical data independence is more difficult to achieve than physical data independence because changes at the logical level can have a direct effect on the information expected by applications and user views.

1.2.6 Applying Data Models and Database Architectures

Understanding different data models and database architectures is useful only when these concepts can be applied to real database requirements. The choice should be based on the nature of the data, relationships, number of users, performance requirements, scalability, and application environment.

The choice of a data model depends on how information is structured and how it is expected to be accessed.

Some general considerations are:

No single data model is suitable for every application. The selected model should support the application's data requirements and expected operations effectively.

Database architecture should also be selected according to the scale and requirements of the application.

SGOU

A centralized architecture may be appropriate when database management is concentrated at a single location. Client–server architecture is useful when multiple client applications need access to a shared database. Three-tier architecture is particularly suitable for web and enterprise applications where presentation, business logic, and database management need to be separated.

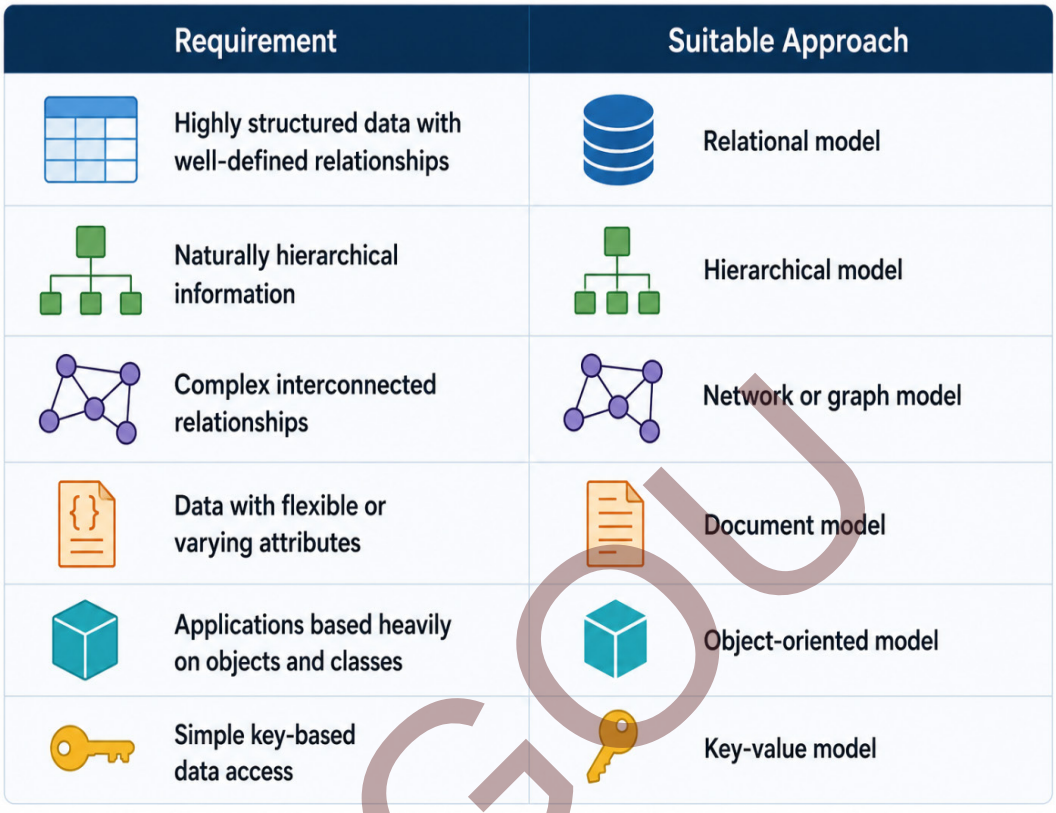


Fig.1.2.10. Choice of a datamodel based on database architecture

For large-scale or geographically distributed applications, distributed or cloud-based database environments may provide greater flexibility and scalability.

Application → Suitable Model → Suitable Architecture

Examples:

- Banking system → Relational model → Three-tier architecture
- Social network → Graph model → Distributed/cloud architecture
- CAD system → Object-oriented model → Client-server architecture

SGOU

Summarised Overview

Data models provide a structured way to represent data, relationships, and constraints in a database. Database schema defines the overall structure of the database, while a database instance represents the actual data stored at a particular time. Database system architectures describe how users, applications, DBMS, and databases are organised and connected. Data abstraction hides unnecessary storage and implementation details through physical, logical, and view levels. Data independence allows changes at one level of database abstraction with minimal impact on higher levels. Together, these concepts provide a foundation for designing database systems that are organised, flexible, maintainable, and suitable for different application requirements.

Assignments

1. Explain the concept and purpose of data models. Describe the major approaches used for representing data in database systems.

2. Explain database schema and database instance. Compare them based on their meaning, nature, and frequency of change with suitable examples.

3. Describe the different levels of database abstraction. Explain the physical, logical, and view levels and their role in simplifying database management.

4. A university plans to develop a database system for managing students, courses, examinations, and academic results. Analyse the most suitable data model and database architecture for this system. Justify your choices.

5. A large e-commerce application needs to improve database performance without affecting existing applications and user views. Analyse how physical and logical data independence can support this requirement, giving suitable examples.

SGOU

Reference

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database system concepts* (7th ed.). McGraw-Hill.

2. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2009). *Database systems: The complete book* (2nd ed.). Pearson.

3. Date, C. J. (2004). *An introduction to database systems* (8th ed.). Pearson.

4. Oppel, A. J. (2018). *Databases demystified* (2nd ed.). McGraw-Hill Education.

Suggested Reading

1. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill.

2. Connolly, T., & Begg, C. (2015). *Database systems: A practical approach to design, implementation, and management* (6th ed.). Pearson.

3. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Pearson.

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

3
UNIT

ER Model

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the key components of the ER model
- describe the different types of attributes with suitable examples
- explain the different types of relationships in the ER model
- differentiate between the various types of keys

Background

To understand the Entity-Relationship (ER) Model, it is important to have a basic understanding of databases and how data is stored. You may already be familiar with organizing information in tables, where rows represent individual records and columns represent the attributes of those records. This table-based structure is a fundamental part of relational databases.

The ER Model builds on this idea by providing a visual framework for representing how different types of data are connected. It can be used to represent real-world elements such as people, products, orders, and their relationships through an ER diagram.

The ER Model supports database design by focusing on three key components:

- Entities – objects or concepts about which information is stored.
- Attributes – properties or characteristics of entities.

SGOU

- Relationships – associations or connections between entities.

You should also have a basic understanding of tables, records, and attributes in a database. These concepts help in understanding how records are uniquely identified using different types of keys.

Keys are important for organizing and managing data effectively in a relational database. Similar to how identifiers such as a phone number or email address can be used to distinguish individuals, database keys help uniquely identify records and maintain data integrity and uniqueness.

In this unit, you will learn about candidate keys, superkeys, primary keys, and composite keys, and understand their roles in the Entity-Relationship (ER) model and relational databases.

Keywords

ER Model, Entity, Attribute, Relationship, Cardinality, Candidate Key, Super Key, Primary Key, Composite Key

Discussion

SGOU

1.3.1 Introduction to ER Model

The Entity-Relationship (ER) Model, introduced by Peter Chen in 1976, is a conceptual framework used for the design and organization of databases. It provides a systematic way to represent data and the relationships among different data elements. The ER Model is widely used in database design because it clearly represents entities, attributes, and relationships.

In the ER Model:

- Entities represent real-world objects or concepts about which information is stored, such as Customer or Product.
- Attributes describe the properties or characteristics of an entity. For example, a Customer entity may have attributes such as Customer_ID, Name, and Email.
- Relationships represent the associations between entities. For example, a Customer may have a relationship with a Product indicating that the customer has purchased the product.

These components are typically represented using an Entity-Relationship Diagram (ERD). In the commonly used Chen notation, entities are represented by rectangles, attributes by ovals, and relationships by diamonds.

The graphical representation helps database designers visualize, organize, and communicate the structure of a database and ensures that the database design accurately represents the required real-world data and relationships.

1.3.2 Definition and Purpose of ER Model

The Entity-Relationship (ER) Model is an abstract, high-level framework used to represent the structure of data using entities, attributes, and relationships. Its main objective is to identify and organize the data requirements of a system in a form that can be transformed into a database schema.

The ER Model simplifies the process of designing complex databases by providing a clear representation of the data and the relationships among different entities. It serves as a foundation for developing an efficient and well-organized database that reflects the requirements of a specific application domain.

Purpose of the ER Model

1. **Visual Representation**

The ER Model provides an intuitive, graphical representation of a database structure. This makes it easier for database designers, developers, and other stakeholders to understand how data is organized and how different entities are related.

2. **Data Organization**

The ER Model helps organize data logically by identifying entities, their attributes, and relationships. A well-designed ER model can help reduce data redundancy and support data consistency during database design.

3. **Foundation for Database Design**

The ER Model serves as an important starting point for database design and development. It provides a structured representation that can be mapped to relational tables and helps ensure that the resulting database accurately represents the required data and relationships.

1.3.3 Components of ER Model

The ER Model comprises three key elements:

1. **Entities**

Entities are objects or concepts that are relevant to the system being modeled. Each entity represents a distinct object and has a set of characteristics. For example, in an educational system, Student and Course are entities, each with its own properties.

SGOU

2. **Attributes**

Attributes are the properties or characteristics that describe an entity. For example, a Student entity may have attributes such as Student_ID, Full_Name, Email_Address, and Birth_Date, which provide information about the student.

3. **Relationships**

Relationships represent the associations or connections between entities. They describe how different entities are related based on real-world interactions. For example, in a library system, a Member and a Book can be connected through a Borrows relationship, indicating that a member borrows a book.

1.3.4 Entities and Attributes

In the Entity-Relationship (ER) Model, entities and attributes play a vital role in organizing and representing data effectively. Entities are the core components or objects within a system that we seek to model. These can be significant items like "Customer," "Product," or "Employee," which we intend to store and manage in the database. Each entity acts as a distinct representation of real-world objects or concepts.

Attributes, on the other hand, are the defining features or qualities that describe each entity in detail. They offer specific information about the entity, such as "CustomerID," "Name," "Address," or "PhoneNumber" for a "Customer" entity. By specifying these characteristics, attributes provide a more complete and detailed understanding of the entity.

Through an understanding of how entities and attributes work together, we can build a well-organized database that accurately reflects the real-world scenarios it is designed to model, ensuring the effective capture and arrangement of all necessary data.

1.3.5 Definition of Entities

An entity refers to an object, individual, location, or concept about which data can be stored. In a database, entities represent the main objects or concepts that need to be managed, such as Student, Employee, Product, or Department. Each entity has a distinct identity and is described by specific attributes.

Entities can be classified into two categories:

- **Strong Entities:** These are independent entities that do not depend on another entity for their identification. For example, Customer in a retail database can be a strong entity because it can be uniquely identified using its own key.
- **Weak Entities:** These are entities that depend on another entity, known as the owner entity, for their identification. For example, Order Item may depend on Order for its complete identification.

In an Entity-Relationship (ER) diagram, entities are typically represented by rectangles. They form the foundation around which attributes and relationships are represented.

SGOU

1.3.6 Types of Attributes

1.3.6.1 Types of Attributes

Attributes are the **properties or characteristics** that provide specific information about an entity. In the ER Model, attributes are represented by **ovals** connected to the entities they describe. Based on their structure and characteristics, attributes can be categorized into the following types:

1. **Simple Attributes**

Simple attributes are indivisible and represent a single value. They cannot be divided into smaller meaningful components.

Examples: Name, Age, and Gender of a PERSON entity.

These attributes provide basic information about an entity.

2. **Composite Attributes**

Composite attributes are made up of multiple components, where each component represents a meaningful part of the attribute.

For example, the Full_Name attribute can be divided into:

Full_Name → First_Name, Middle_Name, Last_Name

Similarly, an Address attribute can consist of:

Address → Street, City, State, Postal_Code

Composite attributes help represent complex information in a structured manner.

3. **Derived Attributes**

Derived attributes are values that can be calculated or obtained from other attributes.

For example, Age can be calculated from the Date_of_Birth attribute. Such attributes are generally not stored directly in the database but are calculated when required.

In an ER diagram, derived attributes are commonly represented using dashed ovals.

4. **Multivalued Attributes**

Multivalued attributes can have multiple values for a single entity.

For example, a PERSON entity may have multiple Phone_Numbers.

In an ER diagram, multivalued attributes are represented using double ovals.

These attributes are useful when an entity can have more than one value for a particular property.

SGOU

1.3.7 Relationships in ER Model

In the Entity-Relationship (ER) Model, relationships represent the associations or connections between two or more entities. They describe how entities are related to or interact with each other within a database.

Relationships help represent the structure and interconnections of data in a real-world system. For example, a STUDENT may enrol in a COURSE, and an EMPLOYEE may work in a DEPARTMENT.

In an ER Diagram, relationships are typically represented using diamonds, with lines connecting the relationship to the associated entities.

Understanding relationships is important for accurately modeling real-world data associations and designing a well-structured database.

1.3.8 Types of Relationships

In the Entity-Relationship (ER) Model, relationships are classified based on their cardinality, which specifies how many instances of one entity can be associated with instances of another entity. The four main types of relationships are

1. **One-to-One (1:1)**

In a one-to-one relationship, each instance of one entity is associated with at most one instance of another entity, and each instance of the second entity is associated with at most one instance of the first entity.

For example, a Person may be assigned one Passport, and each Passport is associated with only one Person.

Example: A student has one birth certificate, and a birth certificate is associated with one student. Therefore, the relationship between STUDENT and BIRTH_CERTIFICATE is one-to-one (1:1).

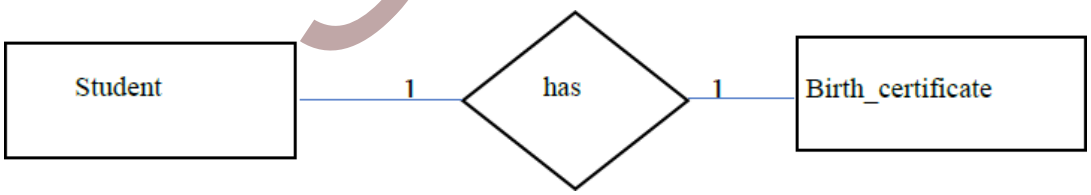


Fig. 1.3.1 One-to-One Relationship

2. **One-to-Many (1:M)**

In a one-to-many relationship, one instance of an entity can be associated with multiple instances of another entity, while each instance of the second entity is associated with only one instance of the first entity.

Example: A student can borrow many books, while each book is assigned to only one student at a time. Therefore, the relationship between STUDENT and BOOK is one-to-many (1:M).

SGOU

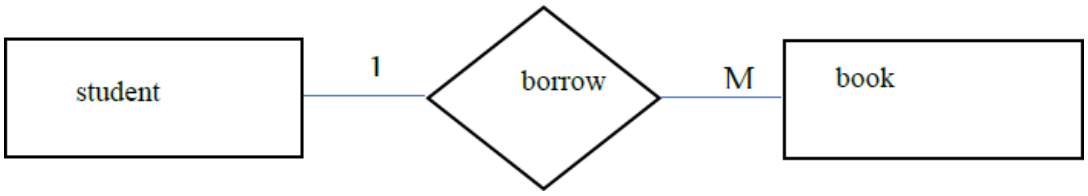


Fig. 1.3.2 One-to-Many Relationship

3. Many-to-One (M:1)

In a many-to-one relationship, multiple instances of one entity can be associated with a single instance of another entity.

Example: Many students can belong to a single department, while a student belongs to only one department at a time. Therefore, the relationship from STUDENT to DEPARTMENT is many-to-one (M:1).

This is the reverse direction of a one-to-many relationship.

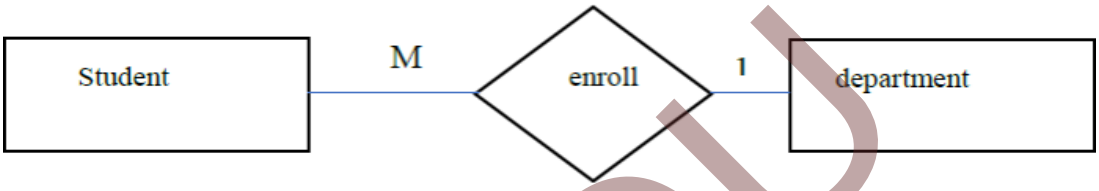


Fig. 1.3.3 Many-to-One Relationship

4. Many-to-Many (M:N)

In a many-to-many relationship, multiple instances of one entity can be associated with multiple instances of another entity.

For example, a student can enrol in multiple courses, and each course can have multiple students. Therefore, the relationship between STUDENT and COURSE is many-to-many (M:N).

Example: A faculty member can teach many students, and a student can be taught by many faculty members. Therefore, the relationship between FACULTY and STUDENT can be represented as a many-to-many (M:N) relationship.

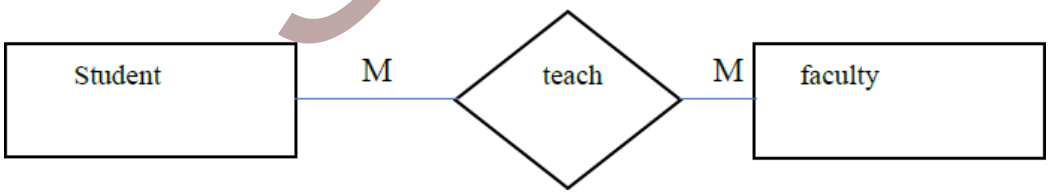


Fig. 1.3.4 Many-to-Many Relationship

1.3.9 Relationship Cardinality

1.3.9.1 Relationship Cardinality

Cardinality defines the numerical relationship between entities. It specifies the minimum and maximum number of instances of one entity that can be associated with instances of another entity.

SGOU

It is commonly expressed using the following forms:

- Mandatory One (1): Each instance of one entity must be associated with exactly one instance of another entity.
- Optional One (0 or 1): Each instance of one entity can be associated with zero or one instance of another entity.
- Mandatory Many (1 or more): Each instance of one entity must be associated with one or more instances of another entity.
- Optional Many (0 or more): Each instance of one entity can be associated with zero, one, or many instances of another entity.

Cardinality helps define the rules and constraints of relationships between entities and ensures that the associations between data are represented correctly.

1.3.9.2 Participation Constraints

Participation constraints determine whether the participation of an entity in a relationship is mandatory or optional. They specify whether every instance of an entity must participate in a relationship or only some instances may participate. These constraints help maintain consistency and integrity in database design.

There are two primary types of participation constraints:

1. **Total Participation (Mandatory)**

In total participation, every instance of an entity must participate in at least one instance of the relationship.

Example: If every Employee must belong to a Department, then the EMPLOYEE entity has total participation in the Assigned_To relationship with the DEPARTMENT entity.

2. **Partial Participation (Optional)**

In partial participation, only some instances of an entity participate in the relationship, while participation is optional for the remaining instances.

Example: If not all Customers are required to place Orders, then the CUSTOMER entity has partial participation in the Places relationship with the ORDER entity.

1.3.10 Keys in ER Model

In the ER Model, keys are used to uniquely identify individual instances of an entity. A key can consist of a single attribute or a combination of multiple attributes. Keys help distinguish one record from another and maintain the integrity of data.

The different types of keys are:

- Candidate Key

SGOU

- Super Key
- Primary Key
- Composite Key
- Foreign Key

Example

Consider a UNIVERSITY entity with the attributes Univ_ID and Univ_Name.

If both Univ_ID and Univ_Name can uniquely identify each university, and each is minimal, they can be considered candidate keys.

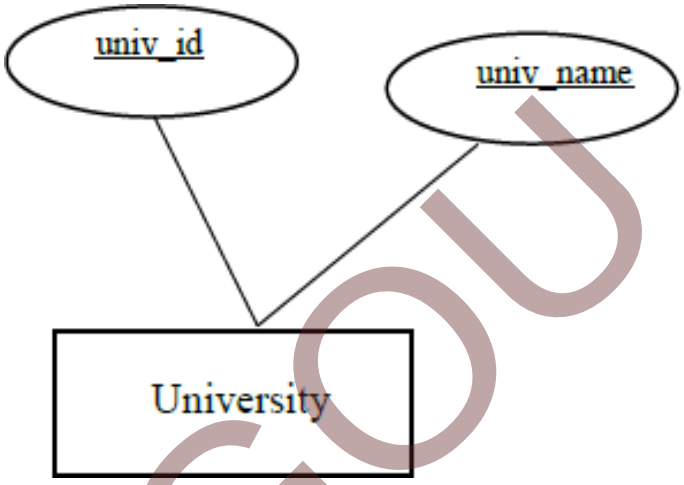


Fig. 1.3.5 ER diagram

For example:

Univ_ID	Univ_Name
U101	Kerala University
U102	Calicut University
U103	Kannur University

Here, Univ_ID can be selected as the primary key because it uniquely identifies each university.

1.3.11Candidate Key

1.3.11.1 Candidate Key

A candidate key consists of one or more attributes that can uniquely identify each record in a table. A candidate key must be unique and minimal, meaning that no unnecessary attribute can be removed while retaining its ability to uniquely identify a record.

A table may have more than one candidate key, and any one of them can be selected as the primary key. The remaining candidate keys are called alternate keys.

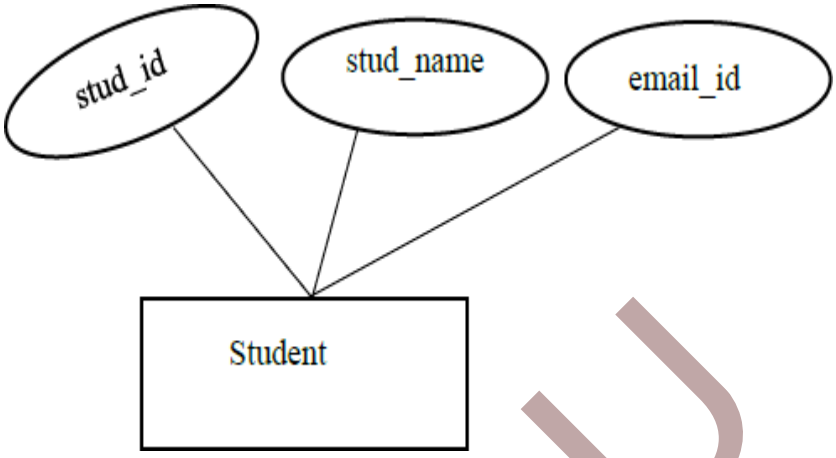
SGOU

Example

Consider a STUDENT table with the following attributes:

STUDENT (Student_ID, Student_Name, Email_ID)

If both Student_ID and Email_ID contain unique values for every student, then both can be candidate keys.



Student_ID	Student_Name	Email_ID
S101	Anu	anu@gmail.com
S102	Arun	arun@gmail.com
S103	Meera	meera@gmail.com

Here:

- Student_ID can uniquely identify each student.
- Email_ID can also uniquely identify each student.
- Therefore, Student_ID and Email_ID are candidate keys.
- If Student_ID is selected as the primary key, **Email_ID** becomes an alternate key.

Note: A candidate key must be **minimal**. For example, (Student_ID, Student_Name) is not a candidate key if Student_ID alone can uniquely identify the student, because Student_Name is unnecessary

1.3.11.2 Super key:

A super key is a set of one or more attributes that can uniquely identify each record (tuple) in a table. A super key may contain additional attributes that are not necessary for maintaining uniqueness.

Example

Consider a STUDENT table:

SGOU

STUDENT (Student_ID, Student_Name, Email_ID)

Student_ID	Student_Name	Email_ID
S101	Anu	anu@gmail.com
S102	Arun	arun@gmail.com
S103	Meera	meera@gmail.com

If Student_ID uniquely identifies each student, then the following are examples of super keys:

- {Student_ID}
- {Student_ID, Student_Name}
- {Student_ID, Email_ID}
- {Student_ID, Student_Name, Email_ID}

All these combinations uniquely identify a student because they contain **Student_ID**.

1.3.11.3 Primary Key

A primary key is a key used in a database to uniquely identify each record (row) in a table. It has the following properties:

1. Uniqueness: The primary key must contain a unique value for each record. No two records can have the same primary key value.
2. Non-null: A primary key cannot contain NULL values. Every record must have a valid value for the primary key.
3. Single or Multiple Attributes: A primary key usually consists of a single attribute (column), but it can also consist of two or more attributes. When multiple attributes are used together as a primary key, it is called a composite key.

For example, in a STUDENT table, Student_ID can be selected as the primary key because each student has a unique Student_ID.

Alternatively, if a primary key consists of Course_ID and Student_ID together, the combination forms a composite primary key.

1.3.11.4 Composite Key

A composite key is a key consisting of two or more attributes (columns) that are combined to uniquely identify a record in a table. It is used when a single attribute alone cannot uniquely identify each record.

For example, in a COURSE_ENROLLMENT table, Student_ID and Course_ID can be combined to form a composite key. Neither attribute alone can uniquely identify an enrollment record, but their combination uniquely identifies each student's enrollment in a particular course.

SGOU

Composite keys are commonly used in many-to-many relationships and in situations where multiple attributes are required to uniquely identify a record.

1.3.11.5 Foreign Key

A foreign key is used to establish a relationship between two tables in a database. It is an attribute in one table that refers to the primary key of another table. This helps represent the connection between records in the two tables.

For example, consider a STUDENT table with Student_ID as its primary key and an ENROLLMENT table that also contains Student_ID. In the ENROLLMENT table, Student_ID acts as a foreign key because it references Student_ID in the STUDENT table.

A foreign key can contain duplicate values because multiple records can refer to the same record in the related table. It may also contain NULL values when permitted by the database constraints.

Foreign keys help maintain referential integrity by ensuring that a foreign key value corresponds to an existing value in the referenced table.

1.3.12 Partial Key in Weak Entities

Consider an entity such as EMPLOYEE. An employee can usually be uniquely identified using Employee_ID. Now consider a weak entity called DEPENDENT, where a dependent may be identified by an attribute such as Dependent_Name.

However, Dependent_Name alone may not uniquely identify a dependent because two different employees may each have a dependent named Anu.

A partial key is an attribute of a weak entity that can distinguish its instances when combined with the primary key of the associated strong entity.

For example:

EMPLOYEE → Employee_ID

DEPENDENT → Dependent_Name

The combination of Employee_ID and Dependent_Name can uniquely identify a dependent. Therefore, Dependent_Name is the partial key of the weak entity.

A partial key cannot uniquely identify a weak entity by itself. It becomes unique when combined with the key of the associated strong entity.

1.3.13 Key Constraints

Two important properties of a key are uniqueness and minimality.

1. **Uniqueness**

A key must have a value that uniquely identifies each record in an entity or table.

SGOU

For example, if Student_ID is a key, two students cannot have the same Student_ID. This ensures that each student can be uniquely identified without ambiguity.

2. Minimality

A key should contain only the attributes necessary to uniquely identify a record. If an attribute can be removed and the remaining attributes can still uniquely identify the record, the original set is not minimal.

For example, if Student_ID alone uniquely identifies a student, using both Student_ID and Name as a key would include an unnecessary attribute. Therefore, Student_ID is the minimal key.

Thus, uniqueness ensures that each record can be distinguished from others, while minimality ensures that no unnecessary attributes are included in the key.

Summarised Overview

The Entity-Relationship (ER) Model is a conceptual framework used to represent and organize data and the relationships among different data elements in a database. It was developed by Peter Chen in 1976 and provides a foundation for database design. The three main components of the ER Model are entities, attributes, and relationships. Entities represent real-world objects or concepts, attributes describe their properties, and relationships represent the connections between entities. These components are commonly represented using Entity-Relationship Diagrams (ERDs).

The ER Model also includes different types of entities, attributes, relationships, cardinality, participation constraints, and keys. Entities may be strong or weak, while attributes may be simple, composite, derived, or multivalued. Relationships may be one-to-one, one-to-many, many-to-one, or many-to-many. Cardinality and participation constraints define how entities are associated with one another.

Keys are important for uniquely identifying records and establishing relationships between tables. The major types of keys include candidate key, super key, primary key, composite key, and foreign key. A partial key is used with a weak entity and becomes unique when combined with the key of its associated strong entity.

Overall, the ER Model helps to organize data, reduce redundancy, maintain data consistency, and provide a structured foundation for designing reliable databases.

SGOU

Assignments

1. What is the role of the Entity-Relationship (ER) Model in database design?
2. Differentiate between strong and weak entities with suitable examples.
3. What is a composite attribute in the ER Model? Explain with a suitable example.
4. Compare one-to-one, one-to-many, and many-to-many relationships in the ER Model with suitable examples.
5. What is a derived attribute in the ER Model? Explain with a suitable example.
6. What is a candidate key in the ER Model? How does it differ from a super key?

Reference

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill.
2. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2009). *Database systems: The complete book* (2nd ed.). Pearson.
3. Connolly, T., & Begg, C. (2015). *Database systems: A practical approach to design, implementation, and management* (6th ed.). Pearson.

Suggested Reading

1. Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.
2. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage Learning.

SGOU

3. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

4 UNIT

ER Model to Database Schema

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the process of mapping an ER model to a relational database schema
- describe how entities and attributes are mapped to tables and columns
- explain how different types of relationships are mapped using foreign keys and separate tables
- describe the mapping of weak entities, including partial keys and composite primary keys
- explain the concepts of generalization and specialization and their mapping to relational tables

SGOU

Background

Imagine you are designing a database for a college. At first, the information may seem simple: students have names, teachers belong to departments, courses have details, and students enrol in courses. However, as the amount of information grows, organizing it properly becomes important.

This is where ER-to-relational mapping becomes important.

An ER model provides a clear representation of the entities, attributes, and relationships in a database. However, a relational database stores information in the form of tables. Therefore, the ER model must be transformed into a suitable set of relational tables while preserving the original information and relationships.

In this unit, you will explore how this transformation takes place. You will learn how entities are mapped to tables, attributes are mapped to columns, and relationships are represented using foreign keys or separate tables. You will also learn how weak entities are handled when they depend on another entity for their identification.

The process also includes generalization and specialization. Real-world entities often have common features as well as special characteristics. For example, an EMPLOYEE may be classified as a MANAGER or an ENGINEER. You will learn how these generalization and specialization concepts can be mapped into relational tables.

By the end of this unit, you will be able to understand how an ER model can be transformed into a structured relational database schema.

Keywords

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

Discussion

SGOU

1.4.1 ER Model to Database Schema

The Entity-Relationship (ER) model is a conceptual approach used to represent the structure of a database. It describes the data requirements of an organisation using entities, attributes, and relationships. Although the ER model is useful for database design, most database management systems store data using the relational model, in which data is organized into tables.

Therefore, after developing an ER model, it is necessary to transform the ER model into a set of relational tables. This process is known as ER-to-relational mapping or mapping the ER model to a relational database schema.

The resulting database schema defines the tables, columns, keys, and relationships required to implement the database. The mapping process must preserve the information represented in the original ER model and maintain the constraints defined during database design.

The major steps involved in mapping an ER model to a relational database schema are:

1. Mapping strong entities to tables

2. Mapping attributes to columns
3. Mapping relationships using foreign keys or separate tables
4. Mapping weak entities
5. Mapping generalization and specialization

1.4.2 Mapping Entities to Tables

An entity in an ER model represents a distinct object or concept about which information needs to be stored. In the relational model, each strong entity is generally represented as a table.

The entity's attributes become the columns of the table, and the entity's key attribute becomes the primary key of the table.

Example

Consider an entity STUDENT with the following attributes:

STUDENT (Student_ID, Student_Name, Email, Age)

The corresponding relational table can be represented as:

Table 1.4.1 Student table

Student_ID	Student_Name	Email	Age
101	Anu	anu@gmail.com	22
102	Arun	arun@gmail.com	23
103	Meera	meera@gmail.com	21

Here, Student_ID uniquely identifies each student and is therefore selected as the primary key.

General Rule

For a strong entity:

- Create one table for the entity.
- Map each simple attribute to a column.
- Select the entity's key attribute as the primary key of the table.
- Represent composite attributes using their individual components.
- Derived attributes are generally not stored if they can be calculated from other stored attributes.

SGOU

For example, if Address is a composite attribute consisting of House_No, Street, City, and PIN_Code, these components can be represented as separate columns in the table.

1.4.3 Mapping Attributes to Columns

The attributes of an entity are mapped to columns in the corresponding relational table. The mapping depends on the type of attribute.

1.Simple Attributes

A simple attribute cannot be divided into smaller meaningful components. It is represented by a single column.

For example:

STUDENT (Student_ID, Name, Age)

Here, Name and Age can be directly mapped to columns.

2. Composite Attributes

A composite attribute consists of multiple components. Instead of creating a separate column for the composite attribute, its individual components are represented as columns.

For example:

Name → First_Name, Middle_Name, Last_Name

The relational table can therefore contain:

Table 1.4.2 Student table with Composite attributes Mapped to Columns

Student_ID	First_Name	Middle_ Name	Last_Name
101	Anu	—	Thomas
102	Arun	Kumar	Nair

3. Multivalued Attributes

A multivalued attribute can have more than one value for a single entity. Since a relational table should normally contain a single value in each cell, a separate table is created for the multivalued attribute.

For example, if a student can have multiple phone numbers:

STUDENT (Student_ID, Student_Name)

and

STUDENT_PHONE (Student_ID, Phone_Number)

Here, Student_ID in STUDENT_PHONE refers to the student, while the combination of

SGOU

Student_ID and Phone_Number can be used as the primary key.

4. Derived Attributes

A derived attribute is calculated from other attributes and generally does not need to be stored separately.

For example, Age can be calculated from Date_of_Birth. Therefore, instead of storing both, the database may store Date_of_Birth and calculate the student's age when required.

1.4.4 Mapping Relationships to Tables

A relationship in an ER model represents an association between two or more entities. When converting an ER model into a relational schema, the relationship is represented using foreign keys or, in some cases, a separate table.

The mapping depends mainly on the cardinality of the relationship.

The common relationship types are:

- One-to-One (1:1)
- One-to-Many (1:N)
- Many-to-Many (M:N)

1.4.4.1 Mapping One-to-One Relationships

A one-to-one (1:1) relationship exists when each instance of one entity is associated with at most one instance of another entity, and each instance of the second entity is associated with at most one instance of the first entity.

When converting a 1:1 relationship from an ER model into relational tables, the primary key of one entity is usually added as a foreign key to the table of the other entity. The foreign key is generally placed in the table where it is more appropriate based on participation constraints and the nature of the relationship.

Consider the following relationship:

Here, Person_ID uniquely identifies a person, while Passport_No uniquely identifies a passport.

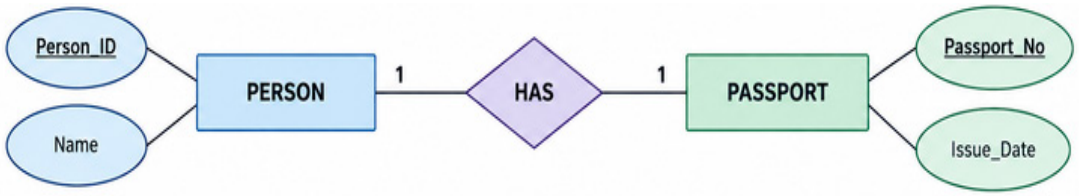


Fig. 1.4.1 ER Diagram

SGOU

To represent the relationship between these two tables, the primary key of PERSON, Person_ID, can be added to the PASSPORT table as a foreign key.

The resulting schema is:

PERSON (Person_ID, Name)

PASSPORT (Passport_No, Issue_Date, Person_ID)

Here:

- Person_ID is the primary key of the PERSON table.
- Passport_No is the primary key of the PASSPORT table.
- Person_ID in the PASSPORT table is a foreign key referencing PERSON(Person_ID).
- A UNIQUE constraint can be applied to PASSPORT.Person_ID so that the same person cannot be associated with multiple passports in this relationship.

Example

Table 1.4.3 Person table

Person_ID	Name
P101	Anu
P102	Arun
P103	Meera
PASSPORT	

Table 1.4.4 Passport table with Person_ID as foreign Key

Passport_No	Issue_Date	Person_ID
PS501	10-02-2025	P101
PS502	15-06-2025	P102
PS503	20-01-2026	P103

The Person_ID value P101 in the PASSPORT table refers to the person with Person_ID = P101 in the PERSON table. Since Person_ID in PASSPORT is unique, the same person cannot appear more than once in this relationship.

1.4.4.2 Mapping One-to-Many Relationships

A one-to-many (1:N) relationship exists when one instance of an entity can be associated with multiple instances of another entity, while each instance of the second entity is associated with only one instance of the first entity.

When mapping a 1:N relationship to a relational database, the primary key of the entity on the one side is added as a foreign key to the table representing the many side. This

SGOU

allows the relationship between the two entities to be represented without creating a separate table for the relationship in most cases.

Example: DEPARTMENT and EMPLOYEE

Consider the following relationship:

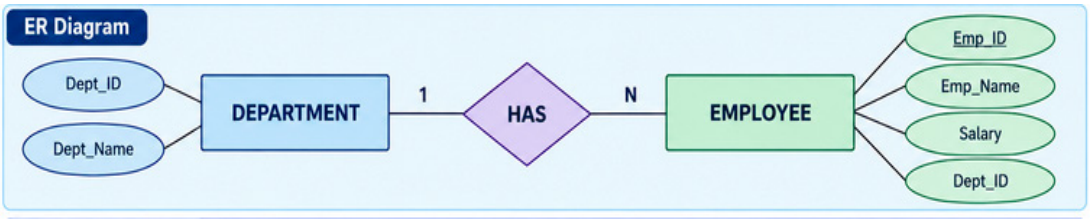


Fig. 1.4.2 ER Diagram

Here, one DEPARTMENT can have many EMPLOYEEs, while each EMPLOYEE belongs to one DEPARTMENT. Therefore, DEPARTMENT is on the one side, and EMPLOYEE is on the many side.

The primary key of DEPARTMENT, Department_ID, is added to the EMPLOYEE table as a foreign key.

The resulting schema is:

DEPARTMENT (Department_ID, Department_Name)

EMPLOYEE (Employee_ID, Employee_Name, Department_ID)

Here:

- Department_ID is the primary key of the DEPARTMENT table.
- Employee_ID is the primary key of the EMPLOYEE table.
- Department_ID in the EMPLOYEE table is a foreign key referencing DEPARTMENT(Department_ID).
- Multiple employees can have the same Department_ID, representing the one-to-many relationship.
- Each employee record contains only one Dept_ID, indicating the department to which that employee belongs.

Example Data

DEPARTMENT

Table 1.4.5 Department table

Department_ID	Department_Name
D01	Computer Science
D02	Mathematics

SGOU

D03	Physics
EMPLOYEE	

Table 1.4.6 Employee Table with Department_ID as Foreign Key.

EMPLOYEE

Employee_ID	Employee_Name	Salary	Department_ID
E101	Anu	50000	D01
E102	Arun	55000	D01
E103	Meera	48000	D02
E104	Rahul	52000	D01

In this example, D01 appears in three records in the EMPLOYEE table. This indicates that three employees belong to the Computer Science department.

The value D01 in each employee record refers to the corresponding Department_ID in the DEPARTMENT table.

1.4.4.3 Mapping Many-to-Many Relationships

A many-to-many (M:N) relationship exists when multiple instances of one entity can be associated with multiple instances of another entity. In this type of relationship, an entity on either side can participate in the relationship with multiple instances of the other entity.

When mapping an M:N relationship to a relational database schema, the relationship is represented using a separate relational table. This table is commonly called a junction table, associative table, or relationship table.

Example: STUDENT and COURSE

Consider the following relationship

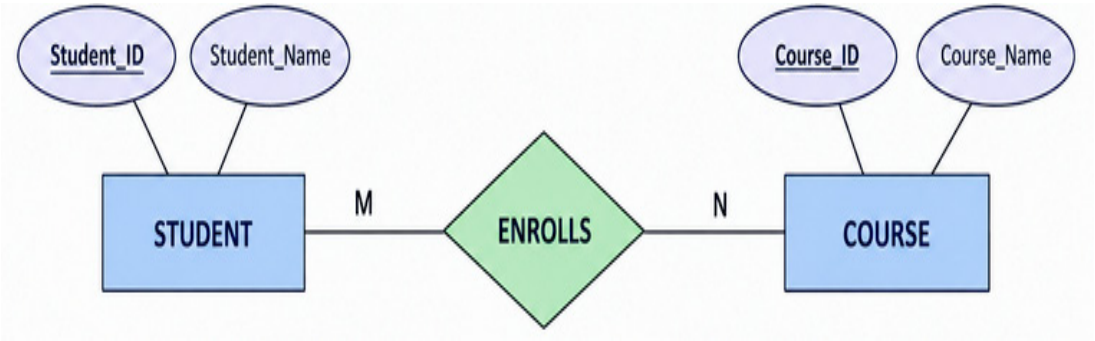


Fig. 1.4.3 M:N Relationship

SGOU

A student can enrol in many courses, and a course can have many students.

Therefore, the relationship is:

STUDENT (M) → ENROLS → (N) COURSE

The entities can initially be represented as:

STUDENT (Student_ID, Student_Name)

COURSE (Course_ID, Course_Name)

Since this is an M:N relationship, a separate table is created to represent the ENROLS relationship:

ENROLLMENT (Student_ID, Course_ID)

Here:

- Student_ID is a foreign key referencing the STUDENT table.
- Course_ID is a foreign key referencing the COURSE table.
- The combination of Student_ID and Course_ID can be used as the composite primary key of the ENROLLMENT table.
- Each record in ENROLLMENT represents one association between a student and a course.

Example Data

STUDENT

Table 1.4.7 Student table

Student_ID	Student_Name
S101	Anu
S102	Arun
S103	Meera

COURSE

Table 1.4.8 Course table

Course_ID	Course_Name
C01	Database Management
C02	Computer Networks
C03	Web Programming

SGOU

ENROLLMENT

Table 1.4.9 Enrollment Table Representing the Many-to Many Relationship

Student_ID	Course_ID
S101	C01
S101	C02
S102	C01
S102	C03
S103	C01

Here, S101 is enrolled in two courses, C01 and C02. Similarly, C01 has multiple students, namely S101, S102, and S103.

Thus, the ENROLLMENT table effectively represents the many-to-many relationship between STUDENT and COURSE.

1.4.5 Mapping Weak Entities

A weak entity is an entity that cannot be uniquely identified using its own attributes alone. It depends on another entity, known as the owner entity or strong entity, for its identification.

A weak entity has a partial key, which can distinguish weak entity instances associated with the same owner entity. However, the partial key alone cannot uniquely identify a record in the database.

When mapping a weak entity to a relational schema, the primary key of the owner entity is included in the weak entity's table. The combination of the owner entity's primary key and the weak entity's partial key forms the composite primary key of the weak entity table.

Example: EMPLOYEE and DEPENDENT

Consider the following ER relationship:

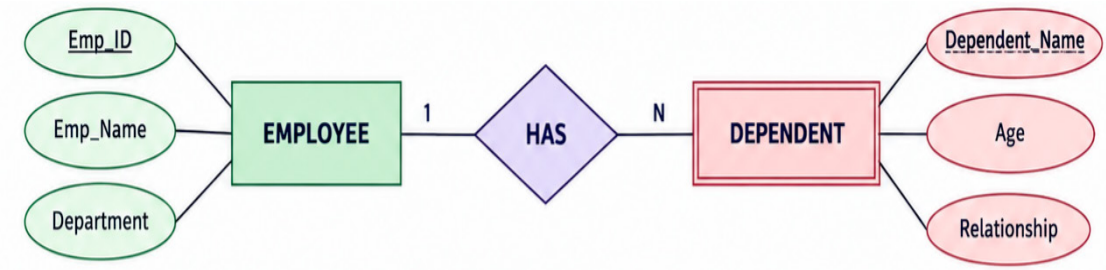


Fig. 1.4.4 Weak Entity

SGOU

An employee may have multiple dependents. A dependent is identified in relation to a particular employee. For example, two different employees may each have a dependent named Rahul. Therefore, the dependent's name alone cannot uniquely identify a dependent.

The entities can be represented as:

EMPLOYEE (Emp_ID, Emp_Name, Department)

DEPENDENT (Dependent_Name, Age, Relationship)

Here:

- Emp_ID uniquely identifies an employee.
- Dependent_Name acts as the partial key of the weak entity.
- DEPENDENT depends on EMPLOYEE for its complete identification.

The relational schema becomes:

EMPLOYEE (Emp_ID, Emp_Name, Department)

DEPENDENT (Emp_ID, Dependent_Name, Age, Relationship)

In the DEPENDENT table:

- Emp_ID is a foreign key referencing EMPLOYEE(Emp_ID).
- Dependent_Name is the partial key of the weak entity.
- (Emp_ID, Dependent_Name) together form the composite primary key of the DEPENDENT table.

Example Data

EMPLOYEE

Table 1.4.10 Employee table

Emp_ID	Emp_Name	Department
E101	Anu	HR
E102	Arun	Finance

DEPENDENT

Table 1.4.11 Dependent Table with Composite Primary Key

Emp_ID	Dependent_Name	Age	Relationship
E101	Rahul	10	Son
E101	Meera	7	Daughter
E102	Rahul	12	Son

SGOU

Notice that Rahul appears for both E101 and E102. Therefore, Dependent_Name alone cannot uniquely identify a dependent.

The combination (Emp_ID, Dependent_Name) uniquely identifies each dependent.

For example:

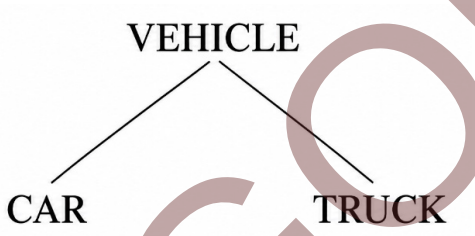
- (E101, Rahul) → Rahul, dependent of employee E101
- (E102, Rahul) → Rahul, dependent of employee E102

Thus, when a weak entity is mapped to a relational schema, the primary key of the owner entity is included in the weak entity table and combined with the partial key to form its composite primary key.

1.4.6 Mapping Generalization and Specialization

1. Generalization

Generalization means combining two or more similar entities into one common entity called a superclass.

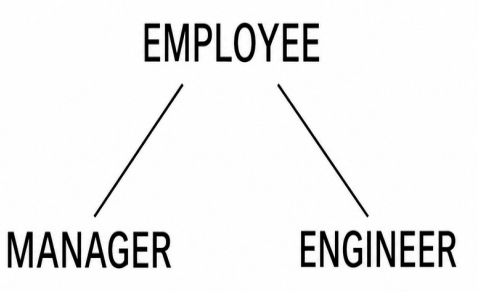


For example, CAR and TRUCK may have common information such as Vehicle_ID, Model, and Price. Therefore, a common superclass called VEHICLE can be created.

Here, VEHICLE is the superclass, while CAR and TRUCK are the subclasses.

2. Specialization

Specialization means dividing one general entity into smaller and more specific entities called subclasses.



For example, an EMPLOYEE can be divided into MANAGER and ENGINEER.

Example of Employee Specialization

Suppose the EMPLOYEE entity has the following common attributes:

SGOU

- Emp_ID
- Name
- Salary

A manager has additional attributes:

- Department
- Bonus

An engineer has additional attributes:

- Engineering_Field
- Skill

The relational tables can be represented as follows.

EMPLOYEE

Table1.4.12 Employee table

Emp_ID	Name	Salary
101	Anu	60000
102	Rahul	55000
103	Suresh	50000

MANAGER

Table 1.4.13 Manager table

Emp_ID	Department	Bonus
101	HR	10000
103	Finance	8000

ENGINEER

Table 1.4.14 Engineer table

Emp_ID	Engineering_Field	Skill
102	Software	Java

Here, Emp_ID connects the superclass table with the subclass tables. It is the primary key of the EMPLOYEE table and also serves as a foreign key in the MANAGER and ENGINEER tables.

For example, Anu (Emp_ID 101) is an employee and also a manager because 101 appears in both the EMPLOYEE and MANAGER tables.

SGOU

Summarised Overview

ER-to-relational mapping is the process of converting an ER model into a set of relational tables while preserving the information, keys, and relationships represented in the model. A strong entity is generally mapped to a table, its attributes become columns, and its key becomes the primary key. Simple attributes are stored directly, composite attributes are represented using their individual components, while multivalued attributes are usually mapped to a separate table. Derived attributes are generally calculated rather than stored.

Relationships are mapped according to their cardinality. In a 1:1 relationship, the primary key of one entity can be added as a foreign key to the other entity's table. In a 1:N relationship, the primary key of the one-side is placed as a foreign key in the many-side table. An M:N relationship requires a separate relationship or junction table containing the keys of the participating entities.

A weak entity depends on an owner entity for identification, so the owner's primary key is included in the weak entity table and, together with the weak entity's partial key, forms a composite primary key.

Generalization and specialization are also important concepts. Generalization combines similar entities into a common superclass, while specialization divides a general entity into more specific subclasses. For example, EMPLOYEE can be specialized into MANAGER and ENGINEER

SGOU

Assignments

1. Explain the process of mapping a strong entity and its attributes from an ER model to a relational database schema. Explain how simple, composite, multivalued, and derived attributes are mapped, with suitable examples.

2. Explain how different types of relationships are mapped into relational tables. Describe the mapping of 1:1, 1:N, and M:N relationships, including the role of primary keys and foreign keys.

3. What is a weak entity? Explain how a weak entity is mapped into a relational database. Discuss the role of the owner entity, partial key, foreign key, and composite primary key with a suitable example.

4. Explain generalization and specialization in ER modeling. Differentiate between them and illustrate how a general entity such as EMPLOYEE can be specialized into MANAGER and ENGINEER.

5. Consider an ER model containing entities, attributes, relationships, a weak entity, and a specialization hierarchy. Explain how the complete ER model can be converted into relational tables. Discuss the important mapping rules and show how primary keys and foreign keys maintain the connections between the tables.

Reference

1. Coronel, Carlos, and Steven Morris. *Database Systems: Design, Implementation, & Management*. 13th ed., Cengage, 2019.

2. Connolly, Thomas, and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. 6th ed., Pearson, 2015.

3. Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed., Pearson, 2008.

Suggested Reading

1. Ramakrishnan, Raghu, and Johannes Gehrke. *Database Management Systems*. 3rd ed., McGraw-Hill, 2003.

2. Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed., McGraw-Hill Education, 2019.

3. Elmasri, Ramez, and Shamkant B. Navathe. *Fundamentals of Database Systems*. 7th ed., Pearson, 2016.

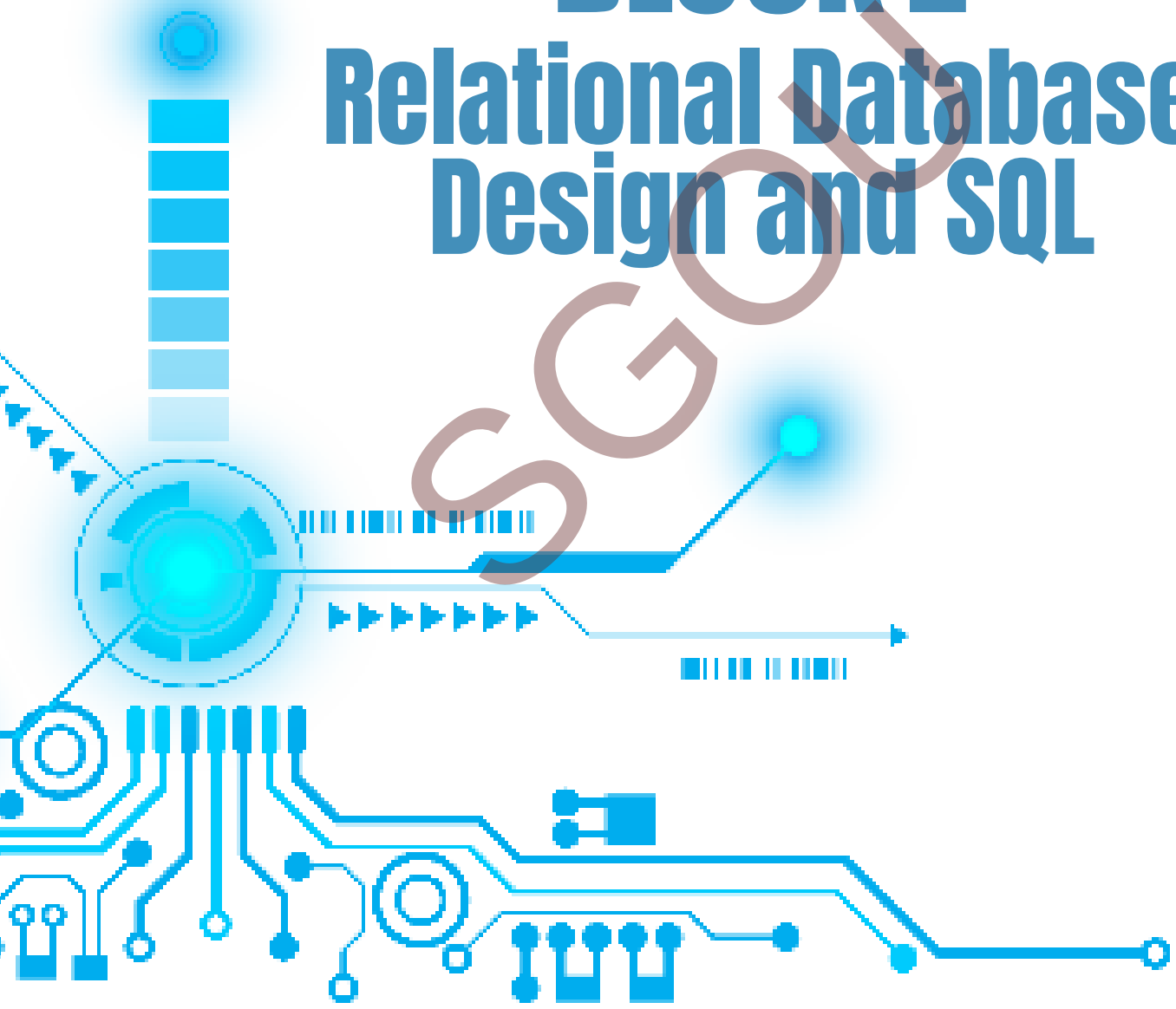
SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

BLOCK 2
Relational Database Design and SQL



SGOU

1 UNIT

Database Normalization

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the concept of functional dependency and its types
- describe the need for database normalization and its benefits
- apply the rules of 1NF, 2NF, and 3NF to organize relational data
- summarize the concept of BCNF, 4NF, and 5NF based on their dependency requirements

Background

The need for studying these normalizations arises from the importance of maintaining well-organized and reliable data in a database. Poorly designed tables may contain repeated information, which can increase storage requirements and make data management difficult. For example, if the same employee address is stored in many records, changing the address may require several records to be updated. Missing even one update can result in inconsistent information. A well-structured database reduces such unnecessary repetition and makes data easier to maintain. These concepts also help identify problems that may occur when adding, modifying, or deleting information. They provide a systematic approach for improving the organisation of database tables. This is particularly useful when designing databases for applications that store large amounts of related information. Studying these concepts also helps students understand how data can be organized efficiently and consistently. Therefore, learning them is necessary for developing reliable and maintainable database systems.

SGOU

Keywords

Data Consistency, Data Integrity, Data Redundancy, Database Design, Functional Dependency, Atomicity, Partial Dependency, Transitive Dependency, Normalization

Discussion

2.1.1 Functional Dependency

A functional dependency describes a relationship between attributes in a database relation. It specifies that the value of one attribute or a set of attributes uniquely determines the value of another attribute. Functional dependency is represented using an arrow (→).

For example, S → T means that the value of S uniquely determines the value of T. Here, S is called the determinant, and T is the dependent attribute.

Consider an EMPLOYEE table with the attributes Emp_Id, Emp_Name, and Emp_Address. Since each employee has a unique employee ID, knowing Emp_Id allows us to determine the corresponding employee name.

Therefore:

Emp_Id → Emp_Name

This means that Emp_Name is functionally dependent on Emp_Id.

2.1.1.1 Types of Functional Dependency

1. **Trivial Functional Dependency**

A functional dependency A → B is trivial when B is a subset of A.

For example:

{Employee_Id, Employee_Name} → Employee_Id

This is trivial because Employee_Id is already part of {Employee_Id, Employee_Name}.

The following are also trivial dependencies:

Employee_Id → Employee_Id

Employee_Name → Employee_Name

SGOU

2. **Non-Trivial Functional Dependency**

A functional dependency A → B is non-trivial when B is not a subset of A.

For example:

ID → Name

Name → DOB

Here, the attributes on the right side are not part of the determinant.

A non-trivial dependency is called a completely non-trivial dependency when A and B have no common attributes.

2.1.2 Armstrong's Axioms

Armstrong's Axioms are a set of rules used to derive functional dependencies from existing dependencies.

2.1.2.1. Augmentation

If S → T, then adding the same attribute set R to both sides does not change the dependency.

SR → TR

Example:

If:

Student_ID → Student_Name

then:

(Student_ID, Course_ID) → (Student_Name, Course_ID)

This makes the axiom easier to apply during normalization problems.

2.1.2.2. Reflexivity

If T is a subset of S, then:

S → T, if a set of attributes S contains all the attributes of another set T, then S functionally determines T.

2.1.2.3. Transitivity

If:

S → T

and

SGOU

$T \rightarrow U$

then:

$S \rightarrow U$, if attribute set S determines attribute set T and T determines attributes set U, then S also determines U. This rule is called the Transitivity Rule.

2.1.3 Advantages of Functional Dependency

1. **Data Accuracy**

Functional dependencies help maintain data consistency by ensuring that dependent attributes are correctly associated with their determinants.

2. **Normalization**

Functional dependencies are important in database normalization. They help identify redundancy and support the decomposition of large tables into well-structured relations.

3. **Efficient Storage**

Normalization based on functional dependencies reduces unnecessary duplication and can therefore reduce storage requirements.

4. **Easy Maintenance**

Well-defined functional dependencies make database modifications easier and reduce the possibility of inconsistencies when data is inserted, updated, or deleted.

2.1.4 Normalization in DBMS

Storing a large amount of information in a single database relation can lead to unnecessary repetition of data. Such data redundancy can create several difficulties in managing the database. The repeated data may make relations unnecessarily large, make data maintenance and modification difficult because many records need to be searched and updated, and result in inefficient use of disk space and other system resources. It can also increase the possibility of errors and inconsistencies in the stored data.

To overcome these problems, relations containing redundant data can be analyzed and decomposed into smaller, simpler, and well-structured relations that satisfy desirable database properties. Normalization is a systematic process of decomposing a relation into smaller relations with fewer attributes while maintaining the required relationships among the data.

2.1.4.1 What is Normalization?

Normalization is the systematic process of organizing data in a relational database by decomposing relations to reduce redundancy and eliminate undesirable dependencies. It is primarily used to minimize data redundancy within a relation or a group of related

SGOU

relations. Normalization also helps eliminate undesirable data modification problems, particularly insertion, update, and deletion anomalies.

During normalization, a large table is divided into smaller, logically organise tables. These tables are then connected through appropriate relationships, such as keys. The different stages of normalization are represented by normal forms, which provide guidelines for reducing redundancy and improving the overall structure of database tables.

2.1.4.2 Why do we need Normalization?

The primary purpose of normalization is to reduce data redundancy and prevent data modification anomalies. If these anomalies are not addressed, the database may contain duplicate or inconsistent information, which can affect data integrity as the database grows.

Normalization provides a systematic set of guidelines for designing well-structured database relations. By applying these guidelines, data can be organized in a way that makes insertion, modification, and deletion operations more reliable and easier to manage.

Improperly designed relations may result in three major types of data modification anomalies:

1. **Insertion Anomaly**
2. **Update Anomaly**
3. **Deletion Anomaly**

2.1.5 Data Modification Anomalies

2.1.5.1 Insertion Anomaly

An insertion anomaly occurs when a new record cannot be added to a relation because some other required information is unavailable. In such a situation, the structure of the table may force the user to provide unrelated or unnecessary data before the new information can be inserted.

Suppose a relation stores Student and Course information together. A new course cannot be added until at least one student registers for it, because student details are mandatory in the same table.

2.1.5.2 Deletion Anomaly

A deletion anomaly occurs when removing a particular record unintentionally causes the loss of other important information. This happens when different types of information are stored together in the same relation and the deletion of one piece of information also removes data that should have been retained.

SGOU

If the only student enrolled in a particular course is deleted, the course information may also be lost from the relation.

2.1.5.3 Update Anomaly

An update anomaly occurs when the same piece of information is stored in multiple records and a change to that information requires several rows to be updated. If all occurrences are not modified correctly, the database may contain inconsistent values for the same information.

If an employee's address is stored in multiple records, changing the address requires updating every occurrence.

2.1.6 First Normal Form (1NF)

A relation is said to be in First Normal Form (1NF) when every attribute contains only a single, atomic value. A table should not contain repeating groups or multiple values within a single cell.

In simple terms, each cell in a table must contain only one value.

2.1.6.1 Rules of First Normal Form

A relation is in 1NF when:

1. Each column contains atomic or single values.
2. Repeating groups are not present.
3. Each row should be uniquely identifiable using a key attribute or set of attributes.
4. Multivalued attributes are not allowed.

Example of a Table Not in 1NF

Consider the following EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

This table is not in 1NF because the EMP_PHONE column contains multiple phone numbers in a single cell. Therefore, the values are not atomic.

SGOU

2.1.6.2 Converting the Table into 1NF

Each phone number can be stored in a separate row.

EMPLOYEE table with seperate rows for each Phone Number

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Now, every cell contains a single value, so the table satisfies the atomicity requirement of 1NF.

2.1.6.3 Methods to Achieve 1NF

Method 1: Remove Repeating Values

Each repeating value is placed in a separate row. The employee information is repeated for each associated phone number. This ensures that every cell contains only one value.

Method 2: Create a Separate Table

The repeating attribute can be separated into another relation.

EMP_DETAILS table: Repeated attributes are seperated into another table.

EMP_ID	EMP_NAME
14	John
20	Harry
12	Sam

EMP_PHONE table set EMP_ID as primary key.

EMP_ID	EMP_PHONE	EMP_STATE
14	7272826385	UP
14	9064738238	UP

SGOU

20	8574783832	Bihar
12	7390372389	Punjab
12	8589830302	Punjab

Here, EMP_ID is the primary key of EMP_DETAILS. In EMP_PHONE, the combination of EMP_ID and EMP_PHONE can be used as the primary key.

This approach avoids storing multiple values in a single cell and provides a better-structured representation of the data.

Method 3: Use Separate Columns

For a fixed and known number of values, multiple columns can be created. For example, if an employee can have a maximum of five skills:

EMP_ID	Skill_1	Skill_2	Skill_3	Skill_4	Skill_5
14	DBMS	C	C++	NULL	NULL
20	JAVA	C	NULL	NULL	NULL
12	DBMS	HTML	VB	MS OFFICE	NULL

Although this satisfies the atomic value requirement, it is generally less flexible because the number of skills is fixed. Queries involving skills may also become difficult because several columns must be checked.

Therefore, separating repeating information into a related table is generally the preferred approach.

2.1.6.4 Anomalies in 1NF

Achieving 1NF removes repeating groups and ensures atomic values, but it does not completely eliminate data redundancy and modification anomalies. The main anomalies are:

1. Insertion anomaly
2. Deletion anomaly
3. Update anomaly

Consider the following ORDER_BOOK relation:

SGOU

ORDER_BOOK table

Order_No	B_Name	Quantity	Price
4253	C	15	175
4253	Database	20	225
4154	IT	30	200
4256	C	50	175
4186	Database	15	225

1. Insertion Anomaly

Suppose information about a new book, Operating System, needs to be added before any order has been placed for it.

Since the combination of Order_No and B_Name uniquely identifies each record, the composite key (Order_No, B_Name) is used as the primary key. Since Order_No cannot be NULL, the book information cannot be inserted without an order number.

For example:

Order_No cannot be NULL

ORDER_BOOK table

Order_No	B_Name	Quantity	Price
4253	C	15	175
4253	Database	20	225
4154	IT	30	200
4256	C	50	175
4186	Database	15	225
NULL	Operating System	30	300

The NULL value cannot be used for the primary key. Therefore, information about the new book cannot be stored independently of an order. This is called an insertion anomaly.

2. Deletion Anomaly

Suppose order number 4154 is cancelled. Deleting this record removes the only stored information about the existence and price of the book IT.

SGOU

Table with deletion anomaly:

ORDER_BOOK table

Order_No	B_Name	Quantity	Price
4253	C	15	175
4253	Database	20	225
4256	C	50	175
4186	Database	15	225

The information about the IT book is lost even though only the cancelled order was intended to be removed. This is called a deletion anomaly.

3. **Update Anomaly**

Suppose the price of the book C changes from 175 to 190. Since C appears in more than one record, every occurrence must be updated.

If one occurrence is not updated, different prices for the same book may exist in the database. This can lead to inconsistent data. Such a problem is called an update anomaly.

NOTE: Although 1NF removes repeating groups and ensures atomicity, it does not eliminate partial dependencies when a relation contains a composite key. Therefore, further normalization through Second Normal Form (2NF) is required.

2.1.7 Second Normal Form (2NF)

Second Normal Form is needed to eliminate partial dependencies and reduce data redundancy by ensuring that every non key attribute depends on the entire primary key.

2.1.7.1 Rules of Second Normal Form

A relation is in Second Normal Form (2NF) if:

1. It is already in First Normal Form (1NF).
2. Every non-key attribute is fully dependent on the entire primary key.
3. There is no partial dependency.

2.1.7.2 Partial Dependency

A partial dependency occurs when a non-key attribute depends on only a part of a composite primary key instead of the complete key.

SGOU

To achieve 2NF, all partial dependencies must be removed.

Example

Consider the following TEACHER table:

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

The combination of TEACHER_ID and SUBJECT forms the primary key. However, TEACHER_AGE depends only on TEACHER_ID, not on the complete primary key. Therefore, the table has a partial dependency and is not in 2NF.

2.1.7.3 Conversion to 2NF

Decompose the table into:

TEACHER_DETAIL

TEACHER_DETAIL table

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT

TEACHER_SUBJECT table

SGOU

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

Now, TEACHER_AGE depends completely on TEACHER_ID, and the partial dependency has been removed. Hence, the relations satisfy 2NF.

2.1.7.4 Anomalies in 2NF

A relation can satisfy 2NF and still have three types of anomalies:

1. **Insertion Anomaly**

A new teacher cannot be added if subjecthas yet been assigned to that teacher, because the student primary key would have to contain a NULL value.

2. **Deletion Anomaly**

Deleting the only subject associated with a teacher may also remove the only stored information about that teacher.

3. **Update Anomaly**

If a teacher's information is repeated in several records, changing that information requires multiple rows to be updated. Failure to update all occurrences can cause inconsistency.

2NF removes partial dependencies, but it may not eliminate all data anomalies. Further normalization is required to achieve a better database structure. This leads to Third Normal Form (3NF).

2.1.8 Third Normal Form (3NF)

Third Normal Form (3NF) is needed to eliminate transitive dependencies, reduce data redundancy, and prevent insertion, update, and deletion anomalies in a relational database.

SGOU

2.1.8.1 Rules of Third Normal Form

A relation is in Third Normal Form (3NF) if:

1. It is already in Second Normal Form (2NF).
2. It has no transitive dependency.
3. Every non-key attribute depends directly on a candidate key.

3NF reduces data redundancy and improves data integrity by removing dependencies between non-key attributes.

2.1.8.2 Transitive Dependency

A transitive dependency occurs when a non-key attribute depends on another non-key attribute rather than directly on the primary key.

For example, if:

A → B and B → C

then **C** is transitively dependent on **A** through **B**.

To achieve 3NF, transitive dependencies must be removed.

2.1.8.3 Conditions for 3NF

For every non-trivial functional dependency X → Y, at least one of the following must hold:

1. X is a super key.
2. Y is a prime attribute, meaning it is part of a candidate key.

Example

Consider the following EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

SGOU

Here, EMP_ID is the candidate key.

The dependencies are:

EMP_ID → EMP_ZIP

EMP_ZIP → EMP_STATE, EMP_CITY

Therefore, EMP_STATE and EMP_CITY depend indirectly on EMP_ID through EMP_ZIP. This is a transitive dependency, so the table is not in 3NF.

2.1.8.4 Conversion to 3NF

The transitive dependency can be removed by decomposing the table into two relations.

EMPLOYEE

EMPLOYEE table

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP

EMPLOYEE_ZIP table

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

In the EMPLOYEE table, the non-key attributes depend directly on EMP_ID. In the EMPLOYEE_ZIP table, EMP_STATE and EMP_CITY depend directly on EMP_ZIP. Thus, the transitive dependency has been removed and the relations satisfy 3NF.

SGOU

2.1.8.5 Anomalies in 3NF

A relation can satisfy 3NF and still contain anomalies caused by multivalued dependencies.

Consider a STAFF relation containing staff members, their equipment, and their languages:

STAFF table

S_Name	Equipment	Language
Anurag	PC	English
Anurag	PC	French
Anurag	Mainframe	English
Anurag	Mainframe	French
Kapil	PC	English
Kapil	PC	French
Kapil	PC	Japanese

The primary key is the combination of S_Name, Equipment, and Language. There is no transitive dependency, so the relation can be in 3NF. However, the independent equipment and language values cause redundancy.

1. **Insertion Anomaly**

If Anurag learns a new language, the new language may have to be combined with each of his equipment records, resulting in multiple insertions.

2. **Deletion Anomaly**

If an equipment record is deleted, information about a staff member's language may also be lost if that record was the only occurrence containing the language.

3. **Update Anomaly**

If a staff member's name or equipment information changes, multiple records may need to be updated. Missing any record can result in inconsistent data.

These anomalies arise from multivalued dependencies. They are addressed by Fourth Normal Form (4NF).

2.1.9 Boyce-Codd Normal Form (BCNF)

Boyce Codd Normal Form (BCNF) is a stricter version of Third Normal Form (3NF). It helps eliminate certain redundancy and anomalies that may remain in a relation even after applying 3NF.

SGOU

2.1.9.1 Definition of BCNF

4. A relation is in Boyce Codd Normal Form if, for every functional dependency $X \rightarrow Y$, X is a super key of the relation.

In simple terms, the left-hand side of every functional dependency must be a candidate key or a super key.

2.1.9.2 Rules of BCNF

A relation satisfies BCNF when:

It is in 3NF.

For every functional dependency $X \rightarrow Y$, X is a super key.

No functional dependency has a non-super key as its determinant.

Example

Consider an organization where employees can work in multiple departments.

EMPLOYEE table

EMP_ID	EMP_ COUNTRY	EMP_DEPT	DEPT_ TYPE	EMP_DEPT_ NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

Functional dependencies are:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

The candidate key is {EMP_ID, EMP_DEPT}.

Why is the Relation Not in BCNF?

EMP_ID determines EMP_COUNTRY, and EMP_DEPT determines DEPT_TYPE and EMP_DEPT_NO. However, neither EMP_ID nor EMP_DEPT alone is a super key.

Therefore, the relation violates the BCNF condition.

SGOU

2.1.9.3 Decomposition into BCNF

The relation can be divided into three tables.

EMP_COUNTRY table

EMP_ID	EMP_COUNTRY
264	India
364	UK

Here, **EMP_ID** is the candidate key and:

EMP_ID → EMP_COUNTRY

EMP_DEPT_DETAILS table

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

Here, **EMP_DEPT** is the candidate key and:

EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

EMP_DEPT_MAPPING table

EMP_ID	EMP_DEPT
264	Designing
264	Testing
364	Stores
364	Developing

Here, **{EMP_ID, EMP_DEPT}** forms the candidate key.

Why is the Decomposed Design in BCNF?

In each table, the determinant of every functional dependency is a super key. Therefore, the three decomposed relations satisfy the requirements of BCNF.

SGOU

BCNF provides a stronger level of normalization than 3NF by ensuring that every determinant is a super key.

2.1.10 Fourth Normal Form (4NF)

Fourth Normal Form (4NF) extends the principles of normalization by addressing **multivalued dependencies**. A relation may satisfy 3NF or BCNF and still contain redundancy when two or more independent multivalued attributes are stored together.

2.1.10.1 Rules of 4NF

A relation is in **Fourth Normal Form (4NF)** if:

1. It is already in **BCNF**.
2. It has no non-trivial multivalued dependency.

A multivalued dependency occurs when one attribute determines a set of values of another attribute independently of the remaining attributes. It is represented as A →→ B.

Example of Multivalued Dependency

Consider the following EMPLOYEE relation:

EMPLOYEE table

Emp_Name	Equipment	Language
Anurag	Personal Computer	English
Anurag	Personal Computer	French
Anurag	Mainframe	English
Anurag	Mainframe	French
Kapil	Personal Computer	English
Kapil	Personal Computer	French
Kapil	Personal Computer	Japanese

For an employee, the equipment and languages are independent of each other. Therefore, storing them in the same relation produces unnecessary combinations and repetition.

The multivalued dependencies are:

Emp_Name →→ Equipment

Emp_Name →→ Language

SGOU

2.1.10.2 Conversion to 4NF

The relation can be decomposed into two relations.

EMP_EQUIP

EMP_EQUIP table

Emp_Name	Equipment
Anurag	Personal Computer
Anurag	Mainframe
Kapil	Personal Computer

EMP_LANG

EMP_LANG table

Emp_Name	Language
Anurag	English
Anurag	French
Kapil	English
Kapil	French
Kapil	Japanese

This decomposition separates the independent multivalued facts and removes unnecessary repetition. The resulting relations satisfy 4NF.

Example 2

Consider the following STUDENT relation:

STUDENT table

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

SGOU

For student 21, courses and hobbies are independent multivalued attributes. Combining them in one relation can create unnecessary combinations.

The relation can therefore be decomposed into:

STUDENT_COURSE

STUDENT_COURSE table

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

STUDENT_HOBBY

STUDENT_HOBBY table

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

The decomposition separates the independent multivalued attributes and reduces redundancy. Thus, the relations can satisfy the requirements of 4NF.

2.1.11 Fifth Normal Form (5NF)

Fifth Normal Form (5NF), also called Project Join Normal Form (PJ/NF), is a higher level of database normalization.

2.1.11.1 Rules of Fifth Normal Form (5NF)

A relation is in 5NF if it is already in Fourth Normal Form (4NF) and has no non-trivial join dependencies. Any decomposition of the relation must also be lossless, meaning that the original relation can be correctly reconstructed by joining the decomposed relations.

2.1.11.2 Join Dependency

A join dependency exists when a relation can be decomposed into two or more smaller relations and then reconstructed by joining those relations without losing or adding incorrect information.

SGOU

5NF removes redundancy caused by such join dependencies. A relation is considered to be in 5NF when every non-trivial join dependency is implied by the candidate keys of the relation.

2.1.11.3 Lossless Decomposition

A decomposition is lossless when joining the decomposed relations produces exactly the original relation. No information is lost and no incorrect additional tuples are generated.

Example

Consider the following FACTORY_COMPONENT_PROJECT relation:

Factory	Component	Project
GM	Engine	MPC
GM	Gear Box	125 A
Honda	Engine	125 A
GM	Engine	125 A

This relation can be decomposed into three relations:

R1: FACTORY_COMPONENT

Factory	Component
GM	Engine
GM	Gear Box
Honda	Engine

R2: COMPONENT_PROJECT

Component	Project
Engine	MPC
Gear Box	125 A
Engine	125 A

R3: FACTORY_PROJECT

Factory	Project
GM	MPC
GM	125 A
Honda	125 A

If the original relation can be reconstructed by joining these three relations without loss of information, the relation satisfies the corresponding join dependency.Since the original relation contains a non-trivial join dependency, that is not implied by candidate

SGOU

keys, it is not in 5NF. Decomposing it into smaller relations removes the redundancy caused by this dependency.

2.1.11.4 Relationship between 4NF and 5NF

A multivalued dependency is a special case of a join dependency. Therefore, a relation in 5NF is also in 4NF. While 4NF mainly deals with multivalued dependencies, 5NF deals with more general join dependencies.

Example of a Relation in 5NF

Consider the following FACTORY_RANKING relation:

Factory_ID	Factory_Name	Position	City
F01	Volvo	17	Mumbai
F11	Honda	22	Chennai
F15	Maruti	26	Gurgaon
F25	GM	37	Chandigarh

Verify the candidate keys. Factory_ID may be a key, but Factory_Name may not always be a candidate key unless uniqueness is guaranteed.

2.1.11.5 Advantages and Disadvantages

5NF reduces redundancy caused by complex join dependencies and helps maintain data consistency.

The main disadvantage is increased database complexity because decomposition may produce more tables and relationships. Queries may also require more joins, which can affect performance.

5NF is mainly concerned with eliminating non-trivial join dependencies. A relation in 5NF cannot be further decomposed into smaller relations without losing information or changing the meaning of the data. It is therefore useful for reducing redundancy in complex relational database designs.

SGOU

Summarised Overview

The material explains functional dependencies as relationships in which one attribute or a set of attributes determines another attribute, and introduces their major types and Armstrong's Axioms. Normalization is presented as a systematic process for decomposing relations to reduce redundancy and prevent insertion, update, and deletion anomalies. First Normal Form ensures atomic values and eliminates repeating groups,

while Second Normal Form removes partial dependencies on composite keys. 3NF further removes transitive dependencies so that non key attributes depend directly on candidate keys. The material then introduces BCNF as a stricter form of 3NF in which every determinant must be a super key. Fourth Normal Form addresses redundancy caused by multivalued dependencies and separates independent multivalued facts into different relations. Fifth Normal Form addresses non trivial join dependencies and requires lossless decomposition of relations. The material provides examples showing how relations can be decomposed into smaller, well structured tables at different normal forms. It also explains how normalization improves data consistency, reduces redundancy, and makes database maintenance more reliable. Overall, the material presents normalization as a progressive process from 1NF to 5NF for designing efficient and well structured relational databases.

Assignments

1. Explain the concept of functional dependency in a relational database. Describe determinant and dependent attributes with suitable examples.
2. Explain the different types of functional dependencies, including trivial, non-trivial, and completely non trivial functional dependencies, with suitable examples.
3. Explain Armstrong's Axioms in detail. Describe reflexivity, augmentation, and transitivity with suitable examples.
4. What is normalization in DBMS? Explain the need for normalization and describe insertion, deletion, and update anomalies with suitable examples.
5. Explain First Normal Form (1NF) in detail. Describe its rules and explain different methods of converting a relation that contains multivalued attributes into 1NF.
6. Explain Second Normal Form (2NF) with a suitable example. What is partial dependency, and how can it be eliminated to achieve 2NF?
7. Explain Third Normal Form (3NF) in detail. Describe transitive dependency and explain how decomposition can be used to convert a relation into 3NF.
8. Explain Boyce-Codd Normal Form (BCNF) and discuss how it differs from 3NF. Illustrate the conversion of a relation that violates BCNF into appropriate relations.

SGOU

9. Explain Fourth Normal Form (4NF) with reference to multivalued dependencies. Describe how decomposition into separate relations helps eliminate redundancy caused by independent multivalued attributes.

10. Explain Fifth Normal Form (5NF), also known as Project Join Normal Form. Describe join dependency and lossless decomposition, and explain how 5NF helps eliminate redundancy caused by complex join dependencies.

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, 7th edition.

Suggested Reading

1. Concept of Database Management, Pratt, Thomson Learning, 5th Edition.

2. Database System Concepts - Silberchatz, Korth and Sudarsan, Fifth Edition, McGraw Hill, 2006

3. The Complete Reference SQL - James R Groff and Paul N Weinberg

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

2
UNIT

SQL Concepts

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the basic concepts and importance of SQLin relational database systems
- identify different SQL data types and select appropriate types for storing database values
- apply basic SQL query structures to retrieve, insert, update, and delete data
- explain different SQL constraints and their role in maintaining data integrity
- demonstrate the use of basic SQL commands with suitable syntax and examples

SGOU

Background

A database is used to store and manage large amounts of information in an organised manner. In a college database, for example, information about learners, courses, teachers, examinations, and marks may be stored in different tables. To work with this stored information, a suitable language is required.Structured Query Language (SQL) provides a standard way to define database structures, insert and retrieve information, modify existing data, and control access to database objects.

Consider an online college admission system. The system may store learner details such as learner ID, name, programme, email address, and admission date. When

an administrator wants to find all learners enrolled in a particular programme, SQL can be used to retrieve the required records. Similarly, SQL can be used to add a newly admitted learner, update an email address, delete an incorrect record, or create a new table. Therefore, understanding SQL is essential for learners who work with relational databases and database applications.

Keywords

SQL, Database, Table, Record, Field, Query, DDL, DML, DQL, DCL, TCL, Data Type, Constraint, Primary Key, Foreign Key, SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, TRUNCATE, COMMIT, ROLLBACK

Discussion

2.2.1 SQL

SQL stands for Structured Query Language. SQL is a standard language used to define, manipulate, retrieve, and control data in relational database management systems.

A relational database stores information in the form of **tables**. A table consists of rows and columns. SQL allows users to create tables, insert data, retrieve data, modify existing data, and delete unwanted data.

2.2.1.1 Need for SQL

SQL is needed to perform different operations on a database. Some important uses of SQL are:

1. Creating database tables.
2. Adding new records.
3. Retrieving required information.
4. Modifying existing records.
5. Removing unwanted records.
6. Defining rules for stored data.

SGOU

7. Defining constraints and rules for maintaining data integrity
8. Maintaining relationships between tables.
9. Controlling user access and permissions.
10. Managing database transactions.

2.2.1.2 Features of SQL

The important features of SQL include:

- It is widely used with relational databases.
- It provides commands for managing data.
- It supports creation and modification of database structures.
- It allows users to retrieve specific information.
- It supports data validation through constraints.
- It provides transaction management facilities such as commit and rollback.
- It supports security and access control mechanisms.
- It can manage large volumes of data efficiently.
- It is supported by many relational database systems.

Examples of database systems that support SQL include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and SQLite.

2.2.2 Basic SQL Commands

SQL commands are categorized into five main groups based on their functionality:

Table 2.2.1 Basic commands in SQL

Category	Full Name	Purpose	Key Commands
DQL	Data Query Language	Retrieve data from the database	SELECT
DDL	Data Definition Language	Define and modify database structure and schema	CREATE, ALTER, DROP, TRUNCATE

SGOU

DML	Data Manipulation Language	Insert, modify, and delete data stored in tables	INSERT, UPDATE, DELETE
DCL	Data Control Language	Manage user privileges and access control	GRANT, REVOKE
TCL	Transaction Control Language	Manage database transactions	COMMIT, ROLLBACK, SAVEPOINT

2.2.3 Data Types in SQL

A data type specifies the kind of value that can be stored in a database column.

For example, a learner's age is normally stored as a number, while the learner's name is stored as text. Selecting a suitable data type helps maintain the correctness of stored data.

Common SQL data types include:

1. Numeric data types
2. Character and string data types
3. Date and time data types
4. Boolean data type

1. Numeric Data Types

Used to store numbers, including integers and floating-point decimals.

Table 2.2.2 Numeric datatypes in SQL

Data Type	Description	Example
INT / INTEGER	Whole numbers (positive, negative, or zero).	42, -105
SMALLINT	Small-range integer values (saves storage).	12
BIGINT	Large-range integer values.	9223372036854775807
DECIMAL(p,s) / NUMERIC	Exact fixed-point numbers, where p represents the total number of digits and s represents the number of digits after the decimal point.	19.99 (via DECIMAL(4,2))

SGOU

FLOAT / REAL	Approximate floating-point numbers for scientific calculations.	3.14159

2. Character and String Data Types

Used to store text, characters, and alphanumeric strings.

Table 2.2.3 Character and string datatypes in SQL

Data Type	Description	Best Used For
CHAR(n)	Fixed-length character string. If the stored value is shorter than n characters, remaining spaces are padded.	Fixed codes (e.g., country codes like 'IN', department codes, ISO codes).
VARCHAR(n)	Variable-length string with a maximum length of n characters	Names, email addresses, titles.
TEXT	Large variable-length string used to store long text values.	Blog posts, product descriptions, reviews.

3. Date & Time Data Types

Date and time data types are used to store temporal information such as dates, times, timestamps, and durations.

Table 2.2.4 Date & Time Data datatypes in SQL

Data Type	Description	Example / Usage
DATE	Stores year, month, and day values in the format **YYYY-MM-DD**.	`2026-08-11` Used for date of birth, admission date, order date
TIME	Stores time-of-day values in the format **HH:MM:SS**.	`14:30:00` Used for class schedules, login time
DATETIME / TIMESTAMP	Stores both date and time values in the format **YYYY-MM-DD HH:MM:SS**. Some DBMSs support **TIMESTAMP WITH TIME ZONE** for storing time zone information.	`2026-08-11 14:30:00` Used for transaction time, record creation time
INTERVAL	Represents a duration or difference between two time values.	`3 DAYS, 2 HOURS` Used for time durations

SGOU

4. Boolean Data Type

Table 2.2.5 Boolean Data Type

Data Type	Description	Example / Usage
BOOLEAN	Stores logical truth values. It accepts TRUE, FALSE, or NULL (unknown).	TRUE / FALSE Used for status indicators such as Active/Inactive

2.2.4 Basic SQL Query Structure

An SQL query is a statement used to retrieve or manipulate information in a database.

The basic structure of a simple SELECT query is:

Syntax:

SELECT * FROM table_name;

SELECT column_name FROM table_name;

SELECT column_name FROM table_name WHERE condition;

Example:

- This displays the name and age of all learners.

```
SELECT Name, Age FROM Learner;
```

- To display all columns:

```
SELECT * FROM Learner;
```

SGOU

Table 2.2.6 SQL Core Clauses

Clause	Purpose	Required?	Example
SELECT	Specifies which columns or calculated results to return.	Yes	SELECT first_name, last_name
FROM	Identifies the table(s) containing the data.	Yes	FROM employees
WHERE	Filters raw rows before any grouping occurs.	Optional	WHERE salary > 50000
GROUP BY	Groups rows sharing a property so aggregate functions (SUM, AVG, COUNT) can be applied.	Optional	GROUP BY department_id

- This query retrieves the names of learners whose course is MCA.

```
SELECT Name FROM Learner WHERE Course = 'MCA';
```

HAVING	Filters groups **after** GROUP BY aggregation has been computed.	Optional	HAVING COUNT(*) > 5
ORDER BY	Sorts the final result set in ascending (ASC) or descending (DESC) order.	Optional	ORDER BY salary DESC
LIMIT	Restricts the total number of rows returned.	Optional	LIMIT 10

2.2.4.1 WHERE Clause

The WHERE clause is used to specify a condition for filtering records from a table. Only rows that satisfy the given condition are included in the result.

Example

```
SELECT *
FROM Learner
WHERE Age > 23;
```

This query retrieves learners whose age is greater than 23.

2.2.4.2 ORDER BY Clause

The ORDER BY clause is used to sort the retrieved records in ascending or descending order based on one or more columns.

Syntax

```
SELECT column_name
FROM table_name
ORDER BY column_name;
```

SGOU

Example

```
SELECT *
FROM Learner
ORDER BY Name;
```

This query retrieves all learner records arranged in ascending alphabetical order of their names.

For descending order:

```
SELECT *
FROM Learner
ORDER BY Age DESC;
```

This query retrieves all learner records arranged in descending order of age, with the oldest learner appearing first.

2.2.4.3 GROUP BY Clause

The GROUP BY clause groups records having the same value in one or more columns. It is commonly used with aggregate functions.

Example

```
SELECT Course
FROM Learner
GROUP BY Course;
```

This displays each distinct course. To count learners in each course, an aggregate function such as COUNT() is required.

2.2.4.4 HAVING Clause

The HAVING clause is used to filter groups created using GROUP BY.

Example

```
SELECT Course
FROM Learner
```

SGOU

```
Difference Between WHERE and HAVINGGROUP BY Course
HAVING Course = 'MCA';
```

Use HAVING with aggregate conditions.Query should include COUNT(*) condition.

Table 2.2.7 Difference Between WHERE and HAVING

WHERE Clause	HAVING Clause
Filters individual rows before grouping is performed.	Filters groups after grouping is performed.
It is generally used before the GROUP BY clause.	It is generally used after the GROUP BY clause.
It is mainly used to filter rows based on column values.	It is mainly used to filter groups based on aggregate functions such as COUNT(), SUM(), and AVG().
Example: WHERE Age > 20	Example: HAVING COUNT(*) > 5

2.2.4.5 LIMIT Clause

The LIMIT clause restricts the number of rows returned in database systems that support it, such as MySQL.

Example

```
SELECT *
FROM Learner
LIMIT 2;
```

This returns up to two records.

2.2.5 Operators in SQL

2.2.5.1 Comparison Operators

Comparison operators are used to create conditions in SQL queries.

Table 2.2.8 Comparison operators

Operator	Meaning
=	Equal to

SGOU

<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

2.2.5.2 Logical Operators

SQL provides logical operators to combine multiple conditions in a query. They are mainly used with the WHERE clause to filter records based on multiple criteria.

Table 2.2.9: Logical operators

AND - Both conditions must be satisfied.	SELECT * FROM Learner WHERE Age > 20 AND Course = 'MCA';
OR- At least one of the conditions should be satisfied.	SELECT * FROM Learner WHERE Course = 'MCA' OR Course = 'MBA';
NOT- NOT reverses a condition.	SELECT * FROM Learner WHERE NOT Course = 'MCA';

2.2.5.3 Membership Operators in SQL

Membership operators are used to check whether a value belongs to a specified set of values. They are useful when a condition needs to compare a column with multiple possible values.

The commonly used membership operators in SQL are:

1. IN
2. NOT IN

1. IN Operator

The IN operator is used to check whether a value matches any value in a specified list.

SGOU

Syntax:

```
SELECT column_name
FROM table_name
WHERE column_name IN (value1, value2, value3);
```

Example:

```
SELECT *
FROM Learner
WHERE Course IN ('MCA', 'MBA');
```

This query retrieves learners whose course is either MCA or MBA.

The IN operator simplifies queries by avoiding the use of multiple OR conditions.

2.2.6 Constraints

A constraint is a rule applied to a table or column to control the values that can be stored in the database.Constraints help maintain data integrity and consistency.

The commonly used SQL constraints are:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

2.2.6.1 NOT NULL Constraint

The NOT NULL constraint ensures that a column cannot contain a NULL value.It prevents NULL values from being inserted into the specified column.

SGOU

Syntax

```
column_namedata_type NOT NULL
```

Example

```
CREATE TABLE Learner (Learner_ID INT NOT NULL,
                      Name VARCHAR(50) NOT NULL
                    );
```

Here, Learner_ID and Name cannot contain NULL values. A record must provide values for these columns before it can be inserted.

2.2.6.2 UNIQUE Constraint

The UNIQUE constraint ensures that all values stored in a column are different. It prevents duplicate values from being entered into the specified column.

Syntax

```
column_namedata_type UNIQUE
```

Example

```
CREATE TABLE Learner (
                      Learner_ID INT UNIQUE,
                     Email VARCHAR(100) UNIQUE
                     );
```

In this example, two records cannot have the same value in the Learner_ID or Email column. Each learner must have a unique learner ID and email address.

2.2.6.3 PRIMARY KEY Constraint

A Primary Key is a constraint used to uniquely identify each record in a table. It ensures that every row has a unique identifier.

A primary key:

SGOU

- Must contain unique values.
- Cannot contain NULL values.
- Identifies each record uniquely.
- A table can have only one primary key.

Syntax

```
column_namedata_type PRIMARY KEY
```

Example

```
CREATE TABLE Learner (
                        Learner_ID INT PRIMARY KEY,
                        Name VARCHAR(50),
                        Course VARCHAR(30)
                      );
```

Here, Learner_ID acts as the primary key and uniquely identifies each learner record in the Learner table.

2.2.6.4 FOREIGN KEY Constraint

A Foreign Key is used to establish a relationship between two tables.

Consider the following tables.

Table 2.2.10 Course Table

Course_ID	Course_Name
C01	MCA
C02	MBA

Table 2.2.11 Learner Table

Learner_ID	Name	Course_ID
101	Anu	C01
102	Rahul	C02

SGOU

Here, Course_ID in the Learner table references Course_ID in the Course table.

Syntax

```
FOREIGN KEY (column_name)
REFERENCES table_name(column_name)
```

Example

```
CREATE TABLE Course (
        Course_IDVARCHAR(10) PRIMARY KEY,
        Course_NameVARCHAR(50)
        );
```

```
CREATE TABLE Learner (
        Learner_ID INT PRIMARY KEY,
        Name VARCHAR(50),
        Course_IDVARCHAR(10),
        FOREIGN KEY (Course_ID)
        REFERENCES Course(Course_ID)
        );
```

The foreign key constraint ensures referential integrity by preventing invalid course IDs from being entered into the Learner table.

2.2.6.5 CHECK Constraint

The CHECK constraint is used to restrict the values that can be stored in a column based on a specified condition. It helps maintain data validity and integrity.

Syntax

```
column_namedata_type CHECK (condition)
```

SGOU

Example

```
CREATE TABLE Learner (
        Learner_ID INT PRIMARY KEY,
        Name VARCHAR(50),
        Age INT CHECK (Age >= 18)
        );
```

Here, the Age column accepts only values that satisfy the condition Age >= 18. Any record with an age value less than 18 will be rejected.

2.2.6.6 DEFAULT Constraint

The DEFAULT constraint automatically assigns a predefined value to a column when no value is provided during data insertion.

Syntax

```
column_namedata_type DEFAULT value
```

Example

```
CREATE TABLE Learner (
        Learner_ID INT PRIMARY KEY,
        Name VARCHAR(50),
        Status VARCHAR(20) DEFAULT 'Active'
        );
```

If no value is supplied for the Status column during insertion, the default value 'Active' is automatically assigned

2.2.7 More About SQL Commands

SQL commands are used to perform different operations on a relational database. Each command has a specific purpose. The commands introduced earlier can be studied in greater detail by considering their syntax and practical examples.

The major SQL commands discussed in this section are:

- CREATE
- ALTER

SGOU

- DROP
- TRUNCATE
- INSERT
- UPDATE
- DELETE
- GRANT
- REVOKE
- COMMIT
- ROLLBACK
- SAVEPOINT

These commands are classified into DDL, DML, DCL, and TCL based on their purpose. DQL is mainly used for retrieving data using SELECT statements.

2.2.7.1 CREATE Command

The CREATE command is a Data Definition Language (DDL) command used to create database objects such as databases, tables, views, and other database structures.

Syntax

```
CREATE TABLE table_name (
        column1 data_type,
        column2 data_type,
        column3 data_type
        );
```

Example

```
CREATE TABLE Learner (
        Learner_ID INT,
        Name VARCHAR(50),
        Course VARCHAR(30),
        Age INT
        );
```

SGOU

This statement creates a table named Learner with four columns:

- Learner_ID – stores learner identification numbers
- Name – stores learner names
- Course – stores course information
- Age – stores learner age

2.2.7.2 ALTER Command

The ALTER command is a Data Definition Language (DDL) command used to modify the structure of an existing database object, such as a table.

It can be used to add, modify, rename, or remove columns depending on the SQL database system.

1. **ADD Column**

The ADD option is used to add a new column to an existing table.

Syntax

```
ALTER TABLE table_name
ADD column_namedata_type;
```

Example

Suppose the Learner table does not contain an Email column.

```
ALTER TABLE Learner
ADD Email VARCHAR(100);
```

This statement adds an Email column to the existing Learner table.Modifying an Existing Column

2. **Modifying an Existing Column**

The MODIFY clause is commonly used in MySQL and Oracle to change the data type or size of an existing column.

SGOU

Syntax

```
ALTER TABLE table_name
MODIFY column_namenew_data_type;
```

Example

```
Suppose the Name column was originally defined as:
Name VARCHAR(30)
To increase its size to 50 characters:
ALTER TABLE Learner
MODIFY Name VARCHAR(50);
```

This changes the definition of the Name column from VARCHAR(30) to VARCHAR(50).

2. RENAME Column

The RENAME COLUMN option is used to change the name of an existing column.

Syntax

```
ALTER TABLE table_name
RENAME COLUMN old_column_name TO new_column_name;
```

Example

```
ALTER TABLE Learner
RENAME COLUMN Name TO Learner_Name;
```

This changes the column name Name to Learner_Name.

3. DROP Column

The DROP COLUMN option is used to remove an existing column from a table.

Syntax

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

SGOU

Example

```
ALTER TABLE Learner
DROP COLUMN Age;
```

This removes the Age column from the Learner table.

2.2.7.3 DROP Command

The DROP COLUMN option is used to remove an existing column from a table.

Syntax

```
DROP TABLE table_name;
```

Example

```
DROP TABLE Learner;
```

This removes the Email column from the Learner table.

Note: DROP should be used carefully because the table and its data are removed.

2.2.7.4 TRUNCATE Command

The TRUNCATE command is a Data Definition Language (DDL) command used to remove all records from a table while preserving the table structure, columns, and constraints.

Syntax

```
TRUNCATE TABLE table_name;
```

Example

```
TRUNCATE TABLE Learner;
```

SGOU

This statement removes all rows from the Learner table. The table structure remains unchanged and the table can be used to store new records.

2.2.7.5 INSERT Command

The INSERT command is a Data Manipulation Language (DML) command used to add new records (rows) to an existing table.

Syntax

```
INSERT INTO table_name
(column1, column2, column3)
VALUES
(value1, value2, value3);
```

Example

```
INSERT INTO Learner
(Learner_ID, Name, Course, Age)
VALUES
(104, 'Arun', 'MCA', 23);
```

This statement inserts a new record into the Learner table with the following values: (104, 'Arun', 'MCA', 23)

2.2.7.6 UPDATE Command

The UPDATE command is a Data Manipulation Language (DML) command used to modify existing records in a table.

Syntax

```
UPDATE table_name
SET column_name = value
WHERE condition;
```

SGOU

The WHERE clause is important when only specific records need to be modified.

Example

Suppose the age of learner with ID **101** needs to be changed to **24**.

```
UPDATE Learner
SET Age = 24
WHERE Learner_ID = 101;
```

This statement updates the Age value to 24 only for the learner whose Learner_ID is 101.

2.2.7.7 DELETE Command

The DELETE command is a Data Manipulation Language (DML) command used to remove existing records from a table.

Syntax

```
DELETE FROM table_name
WHERE condition;
```

The WHERE clause specifies which records should be deleted. It prevents unwanted deletion of records.

Example

To delete the record of learner with ID 104:

```
DELETE FROM Learner
WHERE Learner_ID = 104;
```

Only the record satisfying the condition Learner_ID = 104 is removed from the table.

To delete all records from a table:

```
DELETE FROM Learner;
```

This statement removes all records from the Learner table, but the table structure remains unchanged.

SGOU

Note:The WHERE clause should be used carefully with the DELETE command. If it is omitted, all records in the table will be deleted.

2.2.7.8 GRANT Command

The GRANT command belongs to Data Control Language (DCL). It is used to provide specific privileges or permissions to database users.

Privileges may include permissions to perform operations such as:

- SELECT – retrieve data
- INSERT – add new records
- UPDATE – modify existing records
- DELETE – remove records

Syntax

```
GRANT privilege
ON object
TO user;
```

Example

```
GRANT SELECT
ON Learner
TO user1;
```

This statement gives user1 permission to retrieve data from the Learner table.

2.2.7.9 REVOKE Command

The REVOKE command is a Data Control Language (DCL) command used to remove previously granted privileges or permissions from a database user.

It is used by database administrators to control and manage user access to database objects.

Syntax

```
REVOKE privilege
ON object
FROM user;
```

SGOU

Example

```
REVOKE SELECT
ON Learner
FROM user1;
```

This statement removes the SELECT privilege previously granted to user1 on the Learner table. After revocation, the user can no longer retrieve data from that table.

2.2.7.10 COMMIT Command

The COMMIT command belongs to Transaction Control Language (TCL). It is used to permanently save all changes made during the current transaction.

Once a transaction is committed, the changes cannot normally be undone using the ROLLBACK command.

Syntax

```
COMMIT;
```

Example

```
INSERT INTO Learner
(Learner_ID, Name, Course, Age)
VALUES
(107, 'Neha', 'MCA', 23);
COMMIT;
```

The INSERT statement adds a new learner record temporarily as part of the current transaction. The COMMIT statement permanently saves the insertion in the database.

2.2.7.11 ROLLBACK Command

The ROLLBACK command is a Transaction Control Language (TCL) command used to undo changes made during a transaction that have not yet been committed.

It helps restore the database to its previous state when an error occurs or when the changes are not required.

SGOU

Syntax

```
ROLLBACK;
```

Example

```
UPDATE Learner
                    SET Age = 30
                    WHERE Learner_ID = 101;
                    ROLLBACK;
```

If the update operation has not been committed, the ROLLBACK statement cancels the change and restores the previous value of Age for the learner with Learner_ID = 101.

2.2.7.12 SAVEPOINT Command

The SAVEPOINT command is a Transaction Control Language (TCL) command used to create an intermediate point within a transaction. It allows a transaction to be partially rolled back to a specific point without cancelling the entire transaction.

Syntax

```
SAVEPOINT savepoint_name;
```

To return to a specific savepoint:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

The ROLLBACK TO SAVEPOINT S1 statement cancels the update operation performed after the savepoint while keeping the changes made before S1.

Summarised Overview

SQL is a standard language used to communicate with and manage relational databases. A relational database organises information into tables containing rows and columns. SQL provides commands for creating database structures, inserting and retrieving

SGOU

data, updating existing information, deleting records, controlling user privileges, and managing transactions.

Data types specify the kind of values that can be stored in a column. Common data types include numeric, character, date and time, and Boolean types. SQL queries use clauses such as SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY to retrieve, filter, group, and arrange data.

Constraints are rules used to maintain data integrity and consistencyImportant SQL constraints include NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, and DEFAULT.

SQL commands are classified into different categories:

- **DQL:** SELECT
- **DDL:** CREATE, ALTER, DROP, TRUNCATE
- **DML:** INSERT, UPDATE, DELETE
- **DCL:** GRANT, REVOKE
- **TCL:** COMMIT, ROLLBACK, SAVEPOINT

Important commands include CREATE, ALTER, DROP, TRUNCATE, INSERT, SELECT, UPDATE, DELETE, GRANT, REVOKE, COMMIT, ROLLBACK, and SAVEPOINT. Understanding these commands provides a strong foundation for working with relational databases.

SGOU

Assignments

1. Create a Learner table with suitable columns and data types. Apply at least three different SQL constraints such as PRIMARY KEY, NOT NULL, UNIQUE, CHECK, or DEFAULT.

Write SQL commands to insert five learner records into the table.

Write SQL queries to display:

1. All learner records.
2. Learner names and courses.

3. Learners whose age is greater than 20.

4. Learners arranged in descending order of age.

2. Demonstrate the use of the ALTER TABLE command to:

- Add a new column.
- Modify an existing column.
- Rename a column.
- Remove a column.

3. Explain the differences between DELETE, TRUNCATE, and DROP with suitable examples.

4. Explain different SQL constraints with suitable syntax and examples.

5. Explain DDL, DML, DQL, DCL, and TCL commands with suitable examples.

Reference

1. Coronel, C., & Morris, S. (2023). *Database systems: Design, implementation, & management* (14th ed.). Cengage.

2. MySQL. (2025). *MySQL 8.4 reference manual*. Oracle Corporation.

3. PostgreSQL Global Development Group. (2025). *PostgreSQL documentation*. PostgreSQL.

4. Date, C. J. (2003). *An introduction to database systems* (8th ed.). Pearson.

Suggested Reading

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw Hill.

SGOU

2. Garcia-Molina, H., Ullman, J. D., & Widom, J. (2019). *Database systems: The complete book* (2nd ed.). Pearson.

3. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage.

4. Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

5. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw Hill.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

3
UNIT

Built in Functions

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the use of SQL string functions
- apply SQL numeric functions for calculations
- use SQL aggregate functions to summarize data
- use SQL date and time functions to process dates

Background

The study of SQL string, numeric, aggregate, and date and time functions is important for processing different types of data efficiently within a database. String functions are useful for cleaning, modifying, combining, searching, and formatting textual information stored in database tables. For example, separate first name and last name values can be combined into a complete name using a string function. Numeric functions are used to perform mathematical calculations such as rounding values, finding remainders, calculating powers, and determining square roots. For example, a salary value can be rounded to a required number of decimal places before it is presented in a report. Aggregate functions help summarize information from multiple records and are useful for obtaining totals, averages, counts, minimum values, and maximum values. For example, an organization can calculate the total salary or average salary of its employees using aggregate functions. Date and time functions are used when applications need to store, compare, format, or calculate dates and times. For example, the number of days between an order date and delivery date can be calculated using a date function. Therefore, studying these SQL functions helps transform raw database data into meaningful and useful information for analysis, reporting, and decision-making

SGOU

Keywords

Concatenation, Substring, Padding, Replacement, Formatting, Remainder, Aggregation

Discussion

2.3.1 String Functions in SQL

SQL provides several **string functions** for manipulating and processing textual data stored in database tables. These functions are useful for tasks such as modifying, formatting, searching, comparing, extracting, and cleaning string values. They are commonly used when preparing textual data for analysis, reporting, and presentation.

String functions provide greater flexibility when working with text-based attributes and help transform raw textual data into a suitable format for further processing.

Basic String Functions

2.3.1.1 CONCAT()

The CONCAT() function combines two or more strings into a single string. In the following example, 'John', a space, and 'Doe' are joined together. The space is included as a separate string so that the resulting name is displayed in the usual format.

```
SELECT CONCAT('John', ' ', 'Doe') AS FullName;
```

The first string is John, the second string is a blank space, and the third string is Doe. Therefore, the three strings are combined to produce John Doe.

Output:

John Doe

This function is particularly useful when information stored in separate columns, such as first name and last name, needs to be displayed as a single value.

2.3.1.2 CHAR_LENGTH() / CHARACTER_LENGTH()

The CHAR_LENGTH() function returns the number of characters present in a string. In the following example, the string Hello contains five characters.

SGOU

SELECT CHAR_LENGTH('Hello') AS StringLength;

The function counts each character in the string: H, e, l, l, and o. Therefore, the returned value is 5.

Output:

5

This function can be useful for checking the length of names, addresses, passwords, codes, or other text values. The exact behavior of length functions can vary between SQL database systems, particularly for multibyte or Unicode data, so the DBMS documentation should be consulted when necessary.

2.3.1.3 UPPER() and LOWER()

The UPPER() function converts alphabetic characters into uppercase, whereas LOWER() converts them into lowercase.

SELECT UPPER('hello') AS UpperCase;

SELECT LOWER('HELLO') AS LowerCase;

In the first statement, hello is converted to HELLO. In the second statement, HELLO is converted to hello.

Output:

HELLO

hello

These functions are useful when text needs to be displayed in a consistent case or when data needs to be normalized before comparison.

2.3.1.4 REPLACE()

The REPLACE() function searches for a particular substring within a string and substitutes it with another substring.

SELECT REPLACE('Hello World', 'World', 'SQL') AS UpdatedString;

Here, the function searches for the word World in the string Hello World. Since World is found, it is replaced with SQL. The resulting string is therefore Hello SQL.

Output:

Hello SQL

This function is useful for correcting, cleaning, or transforming text stored in database columns. For example, unwanted characters or outdated text can be replaced with the required value.

SGOU

2.3.1.5 SUBSTRING() / SUBSTR()

The SUBSTRING() function extracts a portion of a string based on a specified starting position and length. The exact syntax can vary between database systems.

SELECT SUBSTRING('Hello World', 1, 5) AS SubStringExample;

In this example, extraction begins at position 1 and continues for 5 characters. The characters are H, e, l, l, and o. Therefore, the extracted substring is Hello.

Output:

Hello

This function is useful when only a specific part of a text value is required, such as extracting a portion of an identification code or a product code.

2.3.1.6 LEFT() and RIGHT()

The LEFT() function extracts a specified number of characters from the beginning of a string, while RIGHT() extracts characters from the end.

SELECT LEFT('Hello World', 5) AS LeftString;

SELECT RIGHT('Hello World', 5) AS RightString;

The first statement extracts the first five characters from Hello World, resulting in Hello. The second statement extracts the last five characters, resulting in World.

Output:

Hello

World

These functions are useful when only the beginning or ending portion of a text value is required. For example, they can be used to extract prefixes, suffixes, or portions of codes.

2.3.1.7 INSTR()

The INSTR() function finds the position of the first occurrence of a specified substring within another string. In systems such as MySQL, the position is counted from 1.

SELECT INSTR('Hello World', 'World') AS SubstringPosition;

In Hello World, the word World begins at the seventh character. Therefore, INSTR() returns 7.

Output:

7

SGOU

This function is useful when a program needs to determine whether a particular word or character exists in a string and where it occurs. The exact function name and behavior may differ between SQL implementations.

2.3.1.8 TRIM()

The TRIM() function removes unwanted characters, commonly spaces, from the beginning and end of a string.

```
SELECT TRIM(' ' FROM '  Hello World  ') AS TrimmedString;
```

The original string contains spaces before Hello and after World. The function removes these leading and trailing spaces while preserving the space between Hello and World.

Output:

Hello World

TRIM() is particularly useful for cleaning data entered by users because unnecessary spaces can cause problems during searching, comparison, or data processing.

2.3.1.9 REVERSE()

The REVERSE() function changes the order of characters in a string so that the last character becomes the first and the first character becomes the last.

```
SELECT REVERSE('Hello') AS ReversedString;
```

The characters in Hello are arranged as H-e-l-l-o. Reversing their order produces o-l-l-e-H.

Output:

olleH

This function can be useful for certain text-processing and pattern-processing operations. It should not, however, be treated as a general password-security mechanism.

2.3.1.10 RTRIM()

The RTRIM() function removes unwanted trailing characters from the right side of a string. When no specific character is supplied, it commonly removes trailing spaces.

```
SELECT RTRIM('SGOU   ');
```

The original string contains additional spaces after the word `SGOU`. `RTRIM()` removes these trailing spaces, leaving only the actual text.

Output:

SGOU

SGOU

This function is useful when text values contain unnecessary spaces at the end, particularly in data imported from external systems.

2.3.1.11 SPACE()

The SPACE() function generates a string containing a specified number of blank spaces.

SELECT SPACE(7);

Here, the argument 7 instructs the function to generate seven spaces.

Output:

Although the spaces are not easily visible in ordinary output, the returned value contains seven blank characters. This function can be useful for certain text-formatting operations.

2.3.2 Advanced String Functions

Advanced string functions provide additional capabilities for processing, formatting, comparing, and manipulating text data.

2.3.2.1 ASCII()

The ASCII() function returns the numeric ASCII code corresponding to a character. In the following example, the character t has an ASCII value of 116.

SELECT ASCII('t');

The function examines the supplied character and returns its corresponding ASCII code.

Output:

116

This function is useful when applications need to work with the numeric representation of characters. ASCII primarily represents standard characters, so it should not be assumed to represent every Unicode character.

2.3.2.2 CONCAT_WS()

CONCAT_WS() means Concatenate With Separator. It combines multiple strings while placing a specified separator between them.

SELECT CONCAT_WS('_', 'SGOU', 'for', 'Learners');

The first argument, _, specifies the separator. The remaining three strings are then joined using this separator. Thus, SGOU, for, and Learners become SGOU_for_Learners.

SGOU

Output:

SGOU_for_Learners

This function is useful when several values need to be combined with a consistent separator, such as an underscore, comma, hyphen, or space.

2.3.2.3 FORMAT()

The FORMAT() function formats a numeric value for presentation according to the syntax and formatting rules supported by the particular database system.

SELECT CONCAT(FORMAT(0.981 * 100, 2), '%') AS PercentageOutput;

First, 0.981 * 100 produces 98.1. The formatting specification N2 requests a number displayed with two decimal places, producing 98.10. The % symbol is then added to the formatted value.

Output:

98.10%

This type of formatting is useful when numerical results need to be presented in a user-friendly form. However, FORMAT() syntax is DBMS-specific, so the query should be adapted when using a different SQL implementation.

2.3.2.4 LOCATE()

The LOCATE() function searches for the position of a substring within another string. Its exact syntax and optional arguments depend on the database system.

SELECT LOCATE('for', 'SGOUforODL', 1);

The function searches for the substring for within SGOUforODL, beginning at the specified starting position. The substring begins at position 5, so the function returns 5.

Output:

5

This function is useful when an application needs to identify where a particular word or sequence of characters occurs within a larger text value.

2.3.2.5 LPAD()

The LPAD() function adds specified characters to the left side of a string until the string reaches the required length.

SELECT LPAD('SGOU', 8, '0');

The original string SGOU contains four characters. The required length is 8, so four additional characters are required. Since 0 is specified as the padding character, four zeros are added to the left.

SGOU

Output:

0000SGOU

LPAD() is useful when values need to follow a fixed-width format, such as adding leading zeros to identification numbers.

2.3.2.6 POSITION()

The POSITION() function determines the position of the first occurrence of a specified substring within another string.

SELECT POSITION('G' IN 'SGOU');

The first occurrence of the character G appears at position 2 in SGOU. Therefore, the function returns 2.

Output:

2

This function is useful when the location of a particular character or substring needs to be determined. Its syntax is standardized differently across some SQL implementations, so DBMS-specific syntax should be checked.

2.3.2.7 REPEAT()

The REPEAT() function creates a new string by repeating another string a specified number of times.

SELECT REPEAT('SGOU', 2);

Here, SGOU is supplied as the string and 2 specifies the number of repetitions. Therefore, the function joins two copies of SGOU.

Output:

SGOUSGOU

This function can be useful for generating repeated text patterns or formatting output.

2.3.2.8 RPAD()

The RPAD() function adds specified characters to the right side of a string until it reaches a specified length.

SELECT RPAD('SGOU', 7, '0');

The word SGOU has four characters, while the required length is seven. Therefore, three 0 characters are added to the right side.

SGOU

Output:

SGOU000

RPAD() can be useful for producing fixed-width text output and formatting values into a consistent length.

2.3.2.9 STRCMP()

The STRCMP() function is used in MySQL to compare two strings. It determines whether the first string is equal to, smaller than, or greater than the second string and returns a corresponding numeric value.

Syntax

STRCMP(string1, string2)

Parameters

The function accepts two required parameters, string1 and string2. These represent the two strings that need to be compared.

Return Values

The STRCMP() function produces one of three possible results:

- It returns 0 when string1 and string2 are equal.
- It returns -1 when string1 is less than string2.
- It returns 1 when string1 is greater than string2.

Thus, STRCMP() provides a simple way to determine the lexical relationship between two strings.

2.3.3 Numeric Functions in SQL

SQL numeric functions are built-in functions used to perform mathematical and numerical operations on data stored in a database. They are useful for calculations, data analysis, financial applications, statistical processing, reporting, and data presentation. These functions allow users to perform operations such as finding absolute values, rounding numbers, calculating remainders, determining powers and square roots, and generating random numbers directly within SQL queries.

2.3.3.1 ABS() – Finding the Absolute Value

The ABS() function returns the absolute value of a number. The absolute value represents the magnitude of a number without considering whether it is positive or negative. Therefore, a negative value is converted into its corresponding positive value, while a positive value remains unchanged.

SGOU

Syntax:

SELECT ABS(number);

Example:

SELECT ABS(-25);

In this example, the input value is -25. The ABS() function removes the negative sign and returns its magnitude, which is 25.

Output:

25

This function is useful when the direction or sign of a numerical difference is not important, such as calculating the absolute difference between two values.

2.3.3.2 CEIL() or CEILING() – Rounding Up

The CEIL() or CEILING() function rounds a number upward to the next integer. If a number contains a fractional part, the result is the smallest integer that is greater than or equal to the original number.

Syntax:

SELECT CEIL(number);

Example:

SELECT CEIL(12.34);

The value 12.34 lies between the integers 12 and 13. Since CEIL() always rounds upward, the result is 13.

Output:

13

This function can be useful when a fractional quantity must be treated as a complete unit, such as determining the number of pages, containers, or resources required.

2.3.3.3 FLOOR() – Rounding Down

The FLOOR() function rounds a number downward to the nearest integer. It returns the greatest integer that is less than or equal to the given number.

Syntax:

SELECT FLOOR(number);

SGOU

Example:

SELECT FLOOR(12.98);

The value 12.98 is between 12 and 13. Since FLOOR() rounds downward, the decimal portion is removed and the result becomes 12.

Output:

12

This function is useful when only the completed whole units of a numerical value need to be considered.

2.3.3.4 ROUND() – Rounding to a Specified Precision

The ROUND() function rounds a numerical value to a specified number of decimal places. It is commonly used in financial calculations, statistical reports, and situations where numerical values need to be presented with a particular level of precision.

Syntax:

SELECT ROUND(number, decimal_places);

Example:

SELECT ROUND(15.6789, 2);

Here, the second argument 2 specifies that the result should contain two decimal places. The original value is 15.6789. When rounded to two decimal places, the result is 15.68.

Output:

15.68

Thus, ROUND() differs from FLOOR() and TRUNCATE() because it considers the following digit when determining the rounded result.

2.3.3.5 TRUNCATE() – Removing Decimal Digits

The TRUNCATE() function removes digits after a specified decimal position without performing rounding. It is useful when only a particular number of decimal places needs to be retained.

Syntax:

SELECT TRUNCATE(number, decimal_places);

Example:

SELECT TRUNCATE(12.98765, 2);

SGOU

The second argument specifies that two decimal places should be retained. The original value is 12.98765. After retaining the first two decimal digits, the remaining digits are simply discarded. Therefore, the result is 12.98.

Output:

12.98

For example, unlike ROUND(), TRUNCATE() does not increase 12.98 to 12.99 even though the next digit is 7.

2.3.3.6 MOD() – Finding the Remainder

The MOD() function returns the remainder obtained after dividing one number by another. It is commonly used in mathematical calculations and in determining whether a number is even or odd.

Syntax:

SELECT MOD(dividend, divisor);

Example:

SELECT MOD(10, 3);

When 10 is divided by 3, the quotient is 3 and the remainder is 1. Therefore, MOD(10, 3) returns 1.

Output:

1

For example, MOD(10, 2) returns 0, indicating that 10 is evenly divisible by 2.

2.3.3.7 POWER() – Calculating Powers

The POWER() function raises a number to a specified power. The first argument represents the base, while the second argument represents the exponent.

Syntax:

SELECT POWER(base, exponent);

Example:

SELECT POWER(2, 3);

This calculation means 2^3, which is equivalent to 2 × 2 × 2. Therefore, the result is 8.

Output:

8

The function can be useful in mathematical, scientific, financial, and statistical calculations, such as calculating compound growth.

SGOU

2.3.3.8 SQRT() – Calculating the Square Root

The SQRT() function returns the square root of a given number. The square root of a number is the value that, when multiplied by itself, produces the original number.

Syntax:

SELECT SQRT(number);

Example:

SELECT SQRT(16);

The square root of 16 is 4 because 4 × 4 = 16. Therefore, the function returns 4.

Output:

4

This function can be useful in mathematical, engineering, geometric, and statistical calculations.

2.3.3.9 EXP() – Calculating Exponential Values

The EXP() function returns the value of the mathematical constant e raised to the specified power. The value of e is approximately 2.71828.

Syntax:

SELECT EXP(number);

Example:

SELECT EXP(1);

The expression represents e^1. Since any number raised to the power of 1 remains the same number, the result is approximately 2.71828.

Output:

2.718281828459045

The EXP() function is useful in mathematical and statistical calculations, including exponential growth and probability-related computations.

2.3.3.10 LOG() – Calculating Logarithms

The LOG() function calculates the logarithm of a number. The exact syntax and default logarithm base depend on the SQL database system. In systems such as MySQL, LOG(number) returns the natural logarithm, while LOG(base, number) can be used to specify a particular base.

SGOU

Syntax:

SELECT LOG(number);

SELECT LOG(base, number);

Example:

SELECT LOG(100);

Here, the natural logarithm of 100 is calculated. Since the natural logarithm uses the mathematical constant e as its base, the result is approximately 4.60517.

Output:

4.605170186

Logarithmic functions are commonly used in scientific calculations, statistical analysis, data modelling, and mathematical applications.

2.3.3.11 RAND() – Generating Random Numbers

The RAND() function generates a pseudo-random floating-point value. In database systems such as MySQL, the result is generally a value greater than or equal to 0 and less than 1.

Syntax:

SELECT RAND();

Example:

SELECT RAND();

Each execution can produce a different random value. For example, one execution might return 0.287372.

Output:

0.287372

The actual value may be different each time the query is executed. RAND() can be useful for simulations, random sampling, testing, and applications that require pseudo-random values. The exact behavior, including whether a seed can be supplied, depends on the SQL database system.

2.3.4 Other Mathematical Functions

2.3.4.1 SIN() – Sine Value

The SIN() function calculates the sine of an angle expressed in radians. It is commonly used in trigonometric calculations and has applications in geometry, physics, engineering, and scientific data analysis where angle-based calculations are required.

SGOU

Example

SELECT SIN(PI()/2) AS SineValue;

Output:

SineValue
1

In this example, PI() returns the value of π. Dividing it by 2 gives π/2 radians, which is equivalent to 90 degrees. The sine of 90 degrees is 1. Therefore, SIN(PI()/2) returns 1.

2.3.4.2 COS() – Cosine Value

The COS() function calculates the cosine of an angle specified in radians. It is useful for performing trigonometric calculations in areas such as geometry, physics, engineering, and scientific analysis.

Example

SELECT COS(PI()) AS CosineValue;

Output:

CosineValue
-1

Here, PI() represents π radians, which is equivalent to 180 degrees. The cosine of 180 degrees is −1. Therefore, COS(PI()) produces −1. Due to floating point precision, some database systems may display a value extremely close to −1.

2.3.4.3 TAN() – Tangent Value

The TAN() function calculates the tangent of an angle expressed in radians. It is useful for trigonometric calculations involving angles, slopes, geometry, physics, and engineering applications.

Example

SELECT TAN(PI()/4) AS TangentValue;

Output:

TangentValue 1

In this example, PI()/4 represents π/4 radians, which is equivalent to 45 degrees. The tangent of 45 degrees is 1. Therefore, TAN(PI()/4) returns a value approximately equal to 1. Depending on the database system and its numerical precision, the output may be displayed as 0.999999999999999 rather than exactly 1.

SGOU

2.3.5 SQL Aggregate Functions

SQL aggregate functions are built-in functions used to summarize and analyze data stored across multiple rows in a table. Instead of returning a separate result for every row, an aggregate function processes a set of records and generally produces a single summarized value. These functions are particularly useful for calculating totals, averages, minimum values, maximum values, and record counts.

Aggregate functions are widely used in data analysis and reporting because they make it easier to identify patterns, compare values, and obtain meaningful summaries from large datasets. They can be combined with clauses such as GROUP BY, HAVING, and ORDER BY to perform more detailed analysis.

Most aggregate functions ignore NULL values during calculations. An important exception is COUNT(*), which counts rows regardless of whether individual columns contain NULL values.

General Syntax

AGGREGATE_FUNCTION(column_name);

For example, consider an Employee table containing employee details and salary information.

SELECT SUM(Salary) AS TotalSalary

FROM Employee;

In this query, SUM() processes the values in the Salary column and calculates their total. The AS TotalSalary clause assigns a meaningful name to the result.

2.3.5.1 Common Aggregate Functions in SQL

The following are some of the most commonly used aggregate functions in SQL.

1. **COUNT()**

The COUNT() function determines the number of records or non-NULL values in a specified set of data. It can be used in different forms depending on what needs to be counted.

COUNT(*) counts every row in the result, including rows containing NULL values.

COUNT(column_name) counts only the rows in which the specified column contains a non NULL value.

COUNT(DISTINCT column_name) counts the number of different non-NULL values in the specified column.

SGOU

Example

```
-- Count all records in the Employee table
SELECT COUNT(*) AS TotalRecords
FROM Employee;
-- Count employees having a salary value
SELECT COUNT(Salary) AS NonNullSalaries
FROM Employee;
-- Count different salary values
SELECT COUNT(DISTINCT Salary) AS UniqueSalaries
FROM Employee;
```

The first query counts every row in the Employee table. Therefore, a row is included even if its Salary value is NULL.

The second query counts only those employees for whom a salary value is available. Rows where Salary is NULL are excluded from this calculation.

The third query counts only distinct salary amounts. If several employees receive the same salary, that salary amount is counted only once. NULL values are not included.

For example, if the Salary column contains the values 30000, 40000, 40000, 50000, and NULL, then COUNT(*) would count all five rows, COUNT(Salary) would return four, and COUNT(DISTINCT Salary) would return three because the distinct non-NULL values are 30000, 40000, and 50000.

2. SUM()

The SUM() function calculates the total of numeric values stored in a column. It adds all non-NULL values and is commonly used for calculating total salaries, sales amounts, expenses, marks, or other numerical quantities.

Example

```
-- Calculate the total salary
SELECT SUM(Salary) AS TotalSalary
FROM Employee;
-- Calculate the sum of distinct salary values
SELECT SUM(DISTINCT Salary) AS DistinctSalarySum
```

SGOU

FROM Employee;

In the first query, SUM(Salary) adds all non NULL salary values in the Employee table. The resulting value represents the total salary amount.

In the second query, SUM(DISTINCT Salary) first considers only unique salary amounts and then adds them. Therefore, if the same salary is received by several employees, that salary is included only once.

For example, if the salary values are 30000, 40000, 40000, and 50000, SUM(Salary) produces 160000, whereas SUM(DISTINCT Salary) produces 120000, because the repeated 40000 is counted only once.

NULL values are ignored by both forms of the SUM() function.

3. AVG()

The AVG() function calculates the arithmetic mean of numeric values in a column. It adds the non-NULL values and divides the total by the number of non-NULL values.

Example

```
-- Calculate the average salary

SELECT AVG(Salary) AS AverageSalary

FROM Employee;

-- Calculate the average of distinct salary values

SELECT AVG(DISTINCT Salary) AS DistinctAvgSalary

FROM Employee;
```

In the first query, AVG(Salary) calculates the average salary using all available non NULL salary values.

In the second query, AVG(DISTINCT Salary) considers each different salary amount only once before calculating the average.

For example, if the salary values are 30000, 40000, 40000, and 50000, the average using AVG(Salary) is:

(30000 + 40000 + 40000 + 50000) ÷ 4 = 40000

When AVG(DISTINCT Salary) is used, the duplicate 40000 is removed:

(30000 + 40000 + 50000) ÷ 3 = 40000

Thus, both results happen to be the same in this example, although the calculation methods are different. NULL values are excluded from both calculations.

SGOU

4. MIN() and MAX()

The MIN() and MAX() functions are used to identify the smallest and largest values in a column, respectively. They are useful when determining values such as the lowest salary, highest salary, earliest date, latest date, minimum marks, or maximum marks.

Example

```
-- Find the highest salary

SELECT MAX(Salary) AS HighestSalary

FROM Employee;

-- Find the lowest salary

SELECT MIN(Salary) AS LowestSalary

FROM Employee;
```

In the first query, MAX(Salary) examines the available salary values and returns the highest salary in the table.

In the second query, MIN(Salary) examines the salary values and returns the lowest salary.

For example, if the salary values are 30000, 40000, 45000, 50000, and NULL, MAX(Salary) returns 50000, while MIN(Salary) returns 30000. The NULL value does not affect either calculation.

Therefore, aggregate functions such as COUNT(), SUM(), AVG(), MIN(), and MAX() provide an efficient way to summarize large amounts of data and obtain useful information from database tables.

2.3.6 MySQL Date and Time Functions

MySQL provides a collection of built-in date and time functions for storing, processing, and manipulating date and time information efficiently. These functions are particularly useful in applications that depend on time-based data, such as transaction systems, scheduling applications, attendance systems, and order management systems. They can be used to store timestamps, perform calculations on dates, extract specific portions of date and time values, and present dates in different formats.

These functions are commonly used to:

- Store and manage dates, times, and timestamps.
- Add or subtract specific time intervals from dates.
- Extract components such as the year, month, day, hour, or minute.
- Format date and time values according to application requirements.

SGOU

Date and Time Data Types

MySQL provides several data types for storing date and time information.

- DATE stores only the date in the form YYYY-MM-DD. It is suitable for values such as a person's date of birth, an order date, or an admission date.
- **DATETIME** stores both the date and the exact time. It is useful when an application needs to record a complete date and time value.
- **TIMESTAMP** stores date and time information and is commonly used to record when a record was created or modified. TIMESTAMP values undergo **UTC/session-time-zone conversion**, while DATETIME generally stores the supplied date/time without that conversion.
- YEAR stores only the year, such as 2025 or 2026.

Different Date and Time Functions

MySQL date and time functions make it easier to retrieve, modify, compare, and format temporal data. The following functions are commonly used for working with dates and times.

1. **NOW()**

The NOW() function returns the current date and time. The result is generally displayed in the format YYYY-MM-DD HH:MI:SS.

Query:

SELECT NOW();

Explanation:

When this query is executed, MySQL retrieves the current date and current time from the database server. For example, if the query is executed on 2 May 2026 at 10:30:45 AM, the result would be similar to 2026-05-02 10:30:45. Thus, NOW() is useful when both the current date and current time are required.

2. **CURDATE()**

The CURDATE() function returns the current date without the time component. The result is normally displayed in YYYY-MM-DD format.

Query:

SELECT CURDATE();

Explanation:

This query returns only the current calendar date. For example, if the current date is 2 May 2026, the result will be 2026-05-02. The function is useful when only the date is required, such as when recording the current order date or today's attendance date.

SGOU

3. CURTIME()

The CURTIME() function returns the current time without the date component. The result is generally displayed in HH:MM:SS format.

Query:

SELECT CURTIME();

Explanation:

If the current time is 10:30:45 AM, the query returns 10:30:45. Therefore, CURTIME() is useful when an application requires only the current time, such as recording the time at which an event occurs.

4. DATE()

The DATE() function extracts only the date portion from a DATETIME or TIMESTAMP value. The time portion is removed from the result.

Example:

Consider a table named Test containing a column called BirthTime, which stores both the date and time of birth.

Query:

SELECT Name, DATE(BirthTime) AS BirthDate

FROM Test;

Explanation:

Suppose the BirthTime value for a student is 2000-05-15 14:30:25. The DATE() function removes the time portion and returns 2000-05-15. Therefore, the query displays the student's name along with only the date of birth. This is useful when the time component is not required for the analysis.

5. EXTRACT()

The EXTRACT() function retrieves a specific component from a date or datetime value. It can be used to obtain values such as the year, month, day, hour, minute, second, week, quarter, or microsecond.

Syntax:

EXTRACT(unit FROM date);

Extracting the Day

Query:

SELECT Name, EXTRACT(DAY FROM BirthTime) AS BirthDay

SGOU

FROM Test;

Explanation:

If the BirthTime value is 2000-05-15 14:30:25, EXTRACT(DAY FROM BirthTime) returns 15. Thus, the query displays the name of the person and the day of the month on which the person was born.

Extracting the Year

Query:

SELECT Name, EXTRACT(YEAR FROM BirthTime) AS BirthYear

FROM Test;

Explanation:

For a BirthTime value such as 2000-05-15 14:30:25, the function extracts 2000. Therefore, the query can be used to display the year of birth separately from the complete date and time value.

Extracting the Seconds

Query:

SELECT Name, EXTRACT(SECOND FROM BirthTime) AS BirthSecond

FROM Test;

Explanation:

If the time portion of BirthTime is 14:30:25, the function extracts 25, which represents the seconds component. This is useful when individual time components are required for analysis or comparison.

6. DATE_ADD()

The DATE_ADD() function adds a specified time interval to a date or DATETIME value. The interval can represent units such as days, months, years, hours, minutes, or seconds.

Syntax:

DATE_ADD(date, INTERVAL expr type);

Adding One Year

Query:

SELECT Name,

DATE_ADD(BirthTime, INTERVAL 1 YEAR) AS BirthTimeModified

FROM Test;

SGOU

Explanation:

If the original BirthTime is 2000-05-15 14:30:25, adding one year produces 2001-05-15 14:30:25. The date and time are retained, while the year is increased by one.

Adding 30 Days

Query:

```
SELECT Name,
    DATE_ADD(BirthTime, INTERVAL 30 DAY) AS BirthDayModified
FROM Test;
```

Explanation:

The function adds 30 days to the existing BirthTime value. For example, if the original date is 2000-05-15, adding 30 days results in 2000-06-14. This type of calculation is useful when determining dates after a specified number of days.

Adding 4 Hours

Query:

```
SELECT Name,
    DATE_ADD(BirthTime, INTERVAL 4 HOUR) AS BirthHourModified
FROM Test;
```

Explanation:

This query adds four hours to the existing date and time. For example, if the original value is 2000-05-15 10:00:00, the resulting value becomes 2000-05-15 14:00:00. The function is useful when calculating future times based on a given datetime value.

7. **DATEDIFF()**

The DATEDIFF() function calculates the difference between two dates and returns the result as the number of days.

Syntax:

```
DATEDIFF(date1, date2);
```

Query:

```
SELECT DATEDIFF('2017-01-13', '2017-01-03') AS DateDiff;
```

Explanation:

In this example, the first date is 2017-01-13 and the second date is 2017-01-03. There are 10 days between these two dates, so the function returns 10. This function is useful for calculating the number of days between events, such as the duration between an

SGOU

order date and a delivery date.

8. DATE_FORMAT()

The DATE_FORMAT() function converts a DATE, DATETIME, or TIMESTAMP value into a specified display format. It uses format specifiers to determine how different components of the date and time should appear.

Syntax:

DATE_FORMAT(date, format);

Here, date represents the date or datetime value to be formatted, while format specifies how the result should be displayed.

Some commonly used format specifiers are:

- **%a**: Abbreviated weekday name, such as Sun or Mon.
- **%b**: Abbreviated month name, such as Jan or Feb.
- **%c**: Numeric month.
- **%D**: Day of the month with an English suffix.
- **%d**: Day of the month as a two-digit number.
- **%e**: Day of the month without leading zero.
- **%f**: Microseconds.
- **%H**: Hour in 24-hour format.
- **%h** or **%I**: Hour in 12-hour format.
- **%i**: Minutes.
- **%j**: Day of the year.
- **%k**: Hour in 24-hour format without leading zero.
- **%l**: Hour in 12-hour format without leading zero.
- **%M**: Full month name.
- **%m**: Numeric month with two digits.
- **%p**: AM or PM.
- **%r**: Time in 12-hour format with AM or PM.
- **%S**: Seconds.
- **%T**: Time in 24-hour format.
- **%U**: Week number, with Sunday as the first day of the week.

SGOU

- **%u**: Week number, with Monday as the first day of the week.
- **%V**: Week number with Sunday as the first day of the week.
- **%v**: Week number with Monday as the first day of the week.
- **%W**: Full weekday name.
- **%w**: Numeric day of the week, where Sunday is 0 and Saturday is 6.
- **%X**: Four-digit year associated with the week where Sunday is the first day.
- **%x**: Four-digit year associated with the week where Monday is the first day.
- **%Y**: Four-digit year.
- **%y**: Two-digit year.

Query:

```
SELECT DATE_FORMAT(
    '2025-04-10 12:34:56',
    '%W, %M %d, %Y'
) AS formatted_date;
```

Explanation:

The original datetime value is 2025-04-10 12:34:56. The format string %W, %M %d, %Y specifies that the output should contain the full weekday name, the full month name, the day, and the four-digit year. Therefore, the result is displayed in a form similar to Thursday, April 10, 2025. The time portion is not displayed because the format string does not include any time-related specifiers.

Best Practices for Date and Time in MySQL

Following appropriate practices when working with date and time values helps maintain consistency, accuracy, and efficient query performance.

- Use an appropriate date format: Use standard formats such as YYYY-MM-DD or YYYY-MM-DD HH:MM:SS when inserting or updating date and time values. This helps MySQL interpret the values correctly.
- Use TIMESTAMP when appropriate for time-zone-based data: TIMESTAMP is useful for values that need time-zone-aware handling, whereas DATETIME stores the date and time value without the same automatic time-zone conversion behavior.
- Filter date ranges carefully: The BETWEEN operator can be used in the WHERE clause to retrieve records within a specified date range.

SGOU

Example:

```
SELECT *

FROM orders

WHERE order_date >= '2025-01-01' AND order_date < '2025-02-01';
```

Explanation:

This query retrieves records whose order_date falls between 1 January 2025 and 31 January 2025. It is useful when selecting records belonging to a particular reporting period.

Avoid unnecessary functions in the WHERE clause: Applying functions such as NOW() or CURDATE() directly to database columns in filtering conditions can affect query performance in some situations. Using suitable fixed or precomputed date values can help the database optimize the query more effectively.

Summarised Overview

The study covers SQL string, numeric, aggregate, and date and time functions used for manipulating, processing, calculating, and presenting database values. String functions help combine, measure, convert, replace, extract, search, format, compare, and clean textual data. Functions such as CONCAT(), UPPER(), LOWER(), REPLACE(), SUBSTRING(), LEFT(), RIGHT(), and TRIM() support common text processing operations. Advanced string functions such as ASCII(), CONCAT_WS(), FORMAT(), LOCATE(), LPAD(), POSITION(), REPEAT(), RPAD(), and STRCMP() provide additional capabilities for formatting, searching, padding, repeating, and comparing strings. Numeric functions are used to perform mathematical operations and support calculations, analysis, reporting, and data presentation. Functions such as ABS(), CEIL(), FLOOR(), ROUND(), TRUNCATE(), and MOD() help perform common numerical operations such as finding absolute values, rounding numbers, removing decimal digits, and calculating remainders. Functions including POWER(), SQRT(), EXP(), and LOG() support mathematical and scientific calculations involving powers, square roots, exponential values, and logarithms. Trigonometric functions such as SIN(), COS(), and TAN() are used for angle-based calculations in applications involving mathematics, geometry, physics, and engineering. Aggregate functions process values from multiple rows and provide summarized results that are useful for data analysis and reporting. The commonly used aggregate functions COUNT(), SUM(), AVG(), MIN(), and MAX() help determine record counts, totals, averages, minimum values, and maximum values. Aggregate functions can also be combined with features such as GROUP BY, HAVING,

SGOU

and ORDER BY for more detailed analysis of database records. MySQL date and time functions support the storage, processing, manipulation, comparison, and formatting of temporal information. Functions such as NOW(), CURDATE(), CURTIME(), DATE(), and EXTRACT() help retrieve the current date and time or obtain specific components from date and time values. Functions such as DATE_ADD() and DATEDIFF() support date calculations, while DATE_FORMAT() allows dates and times to be displayed according to specified formatting requirements. Overall, SQL functions provide essential tools for transforming, calculating, analyzing, and presenting database information efficiently in applications and reports.

Assignments

- Explain the purpose and importance of **string functions in SQL**. Describe how string functions can be used for manipulating, formatting, searching, extracting, comparing, and cleaning textual data with suitable examples.
- Explain the commonly used basic string functions in SQL, including CONCAT(), CHAR_LENGTH(), UPPER(), LOWER(), REPLACE(), and SUBSTRING(), with suitable SQL statements and outputs.
- Describe the functions LEFT(), RIGHT(), INSTR(), TRIM(), REVERSE(), RTRIM(), and SPACE(). Explain their uses with appropriate examples.
- Explain the purpose and working of the advanced string functions ASCII(), CONCAT_WS(), FORMAT(), LOCATE(), and POSITION(). Illustrate each function with a suitable SQL example.
- Explain the functions LPAD(), RPAD(), REPEAT(), and STRCMP() in SQL. Describe their syntax, working, and applications with suitable examples.
- What are numeric functions in SQL? Explain the purpose and working of ABS(), CEIL(), FLOOR(), ROUND(), TRUNCATE(), and MOD() with suitable examples.
- Explain the mathematical functions POWER(), SQRT(), EXP(), LOG(), and RAND() in SQL. Discuss their applications and demonstrate their use with appropriate SQL queries.
- Explain the trigonometric functions SIN(), COS(), and TAN() in SQL. Describe how angles are represented and illustrate each function using suitable examples.

SGOU

- What are SQL aggregate functions? Explain COUNT(), SUM(), AVG(), MIN(), and MAX() with suitable examples, including the handling of NULL and DISTINCT values.
- Explain the major MySQL date and time functions used for retrieving, extracting, modifying, calculating, and formatting date and time values. Discuss NOW(), CURDATE(), CURTIME(), DATE(), EXTRACT(), DATE_ADD(), DATEDIFF(), and DATE_FORMAT() with suitable examples.

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, 7th edition.

Suggested Reading

1. Concept of Database Management, Pratt, Thomson Learning, 5th Edition.
2. Database System Concepts - Silberchatz, Korth and Sudarsan, Fifth Edition, McGraw Hill, 2006
3. The Complete Reference SQL - James R Groff and Paul N Weinberg

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

4
UNIT

Views

Learning Outcomes

After completing this unit, the learner will be able to:

- familiarize themselves with the concept and purpose of SQL views in a database
- explain the advantages and disadvantages of using views
- apply SQL statements to create, retrieve data from, modify, and delete views
- describe the rules for updating views and the use of TCL commands with updatable views

SGOU

Background

The study of SQL views is important because database users often require only a specific portion of the information stored in a database. In practical database systems, providing every user with direct access to complete tables may expose unnecessary or sensitive information. To address this requirement, views can provide users with a customized representation of the required data while hiding irrelevant details. For example, consider a university database that contains a StudentDetails table with student ID, name, address, phone number, date of birth, and other personal information. A faculty member may need only the student name and address for academic communication and may not require access to the other personal details. In such a situation, a view can be created to display only the required columns and records. Similarly, information from student and marks tables can be combined through a view so that faculty members can easily

access academic information without repeatedly writing complex queries. Views can therefore simplify database access and support controlled access to data while providing a consistent representation of required information. Hence, studying SQL views helps learners understand how database information can be selectively presented and managed according to the requirements of different users.

Keywords

Query Simplification, Updatable Views, View Management, Transaction Control ,SAVEPOINT, ROLLBACK, COMMIT

Discussion

2.4.1 What is a View?

A view is a virtual table whose contents are defined by a SQL query over one or more database tables or other views.The view definition is stored in the database, while the view generally does not store a separate copy of the underlying data. Unlike a physical table, a view generally does not store a separate copy of the data. When the view is queried, the DBMS uses its definition to retrieve the required data from the underlying objects.

To a database user, a view appears similar to a regular table because it has a name, columns, and rows of data. SQL provides this table-like appearanceby assigning a name to the view and storing its query definition in the database. When the view is accessed, the DBMS uses this definition to retrieve the required information from the underlying tables.

Views are particularly useful for controlling access to database information. They can restrict users from accessing specific rows or columns of a table. By creating appropriate views and permissions, a database administrator can provide users with access to only the information they are authorized to see.

Views can also simplify frequently used queries. For example, information obtained by joining the Student and Marks tables can be presented through a single view, allowing users to retrieve the required information without repeatedly writing the complete join query.

Thus, views provide a simplified and controlled way of presenting data while keeping the underlying database structure separate from the user's view of the information.

SGOU

2.4.1.1 Advantages of Views

1. **Security**

Views improve database security by allowing users to access only the information they are authorized to see. A user can be given permission to access a specific view instead of being given direct access to the underlying tables. For example, a view can expose selected columns while hiding sensitive information from unauthorized users.

2. **Query Simplicity**

A view can combine information from several tables and present the result as a single virtual table. This simplifies complex queries because users can query the view instead of repeatedly writing queries involving multiple tables and join conditions.

3. **Structural Simplicity**

Views can provide users with a customized representation of the database structure. Instead of exposing the complete database structure, a view can present only the columns, rows, and derived information relevant to a particular user or application.

4. **Consistency**

Views can provide a consistent representation of database information even when changes are made to the underlying database structure. If the source tables are reorganized or their structure is modified, an appropriately defined view can continue to provide users with a familiar representation of the required data.

5. **Data Integrity**

When an updatable view is used to modify data, the DBMS can enforce applicable constraints defined on the underlying tables. This helps ensure that the data entered or modified through the view remains consistent with the database rules.

6. **Logical Data Independence**

Views can provide a degree of logical data independence by insulating applications from some changes in the underlying database structure. Without views, an application may need to directly depend on the structure of the base tables. By using views, applications can work with a logical representation of the required data, reducing their dependence on the detailed structure of the underlying tables.

2.4.1.2 Disadvantages of Views

1. **Performance**

A view provides the appearance of a table, but the DBMS generally has to process the query that defines the view against its underlying tables. If the view is based on

SGOU

a complex query involving multiple tables, joins, aggregations, or other operations, retrieving data through the view may require considerable processing time.

2. **Update Restrictions**

Updating data through a view is subject to certain restrictions. When a user attempts to modify data through a view, the DBMS must translate the operation into corresponding changes to the underlying tables. This is generally possible for simple views, but views based on complex queries, multiple tables, aggregation, or other operations may not support direct updates and may therefore be read only.

2.4.2 Creating and Using SQL Views

A SQL View is a virtual table generated from the result of a SELECT statement. Unlike a physical table, a view normally does not store a separate copy of the underlying data. Instead, it presents data retrieved from one or more underlying tables.

Views are useful for simplifying complex queries, controlling access to data, and presenting information in a customized and easy-to-understand format.

2.4.2.1 Creating Views in SQL

A view can be created using the CREATE VIEW statement. The query defining a view may retrieve data from a single table or from multiple tables, depending on the information that needs to be presented.

Syntax:

CREATE VIEW view_name AS

SELECT column1, column2, ...

FROM table_name;

Example: Creating a Simple View

Consider a database containing a StudentDetails table with information such as student ID, name, address, phone number, and date of birth. Suppose a user needs to access only the student's name and address.

A view can be created to display only these required columns.

Query:

CREATE VIEW StudentView AS

SELECT Name, Address

FROM StudentDetails;

SGOU

Retrieving Data from a View

Once the view has been created, it can be queried like a table.

SELECT *

FROM StudentView;

The query retrieves all records and columns available through the StudentView.

Using WHERE

Syntax:

CREATE VIEW view_name AS

SELECT column1, column2.....

FROM table_name

WHERE condition;

Here:

- view_name represents the name assigned to the view,
- column1, column2, ... specify the columns to be included in the view.
- table_name identifies the table from which the data is obtained
- condition specifies an optional condition used to select the required rows.

Note: The WHERE clause is optional. A view can be created without a condition if all rows satisfying the SELECT query are to be made available through the view.

Example 1: Creating a Simple View from a Single Table

Consider a StudentDetailstable containing information about students. Suppose a view is required to display only the name and address of students whose S_ID is less than 5.

Query:

CREATE VIEW DetailsView AS

SELECT NAME, ADDRESS

FROM StudentDetails

WHERE S_ID < 5;

The above statement creates a view named DetailsView. It selects the NAME and ADDRESS columns from StudentDetails and includes only records for which S_ID is less than 5.

SGOU

Retrieving Data from the View

Once the view has been created, data available through the view can be retrieved using a SELECT statement.

SELECT * FROM DetailsView;

This query displays all the rows and columns available through DetailsView.

2.4.2.2 Creating a View from Multiple Tables

A view can also be created by combining information from multiple tables. This is useful when the required information is distributed across related tables.

For example, suppose StudentDetails contains student information and StudentMarks contains examination marks. Both tables have a common S_ID that uniquely identifies each student.

Query:

```
CREATE VIEW MarksView AS
SELECT
StudentDetails.S_ID,
StudentDetails.Name,
StudentDetails.Address,
StudentMarks.Marks
FROM StudentDetails
JOIN StudentMarks
    ON StudentDetails.S_ID = StudentMarks.S_ID;
```

The above statement creates a view named **MarksView** by combining the S_ID, Name, and Address columns from StudentDetails with the Marks column from StudentMarks.

The JOIN condition matches corresponding records from the two tables using the student's unique S_ID.

Retrieving Data from the View

The contents of the view can be displayed using:

SELECT * FROM MarksView;

This query retrieves the combined student and marks information available through MarksView.

SGOU

2.4.3 Managing Views

SQL provides several operations for creating, examining, modifying, and removing views. Some commonly used operations are described below.

2.4.3.1 Listing All Views in a Database

All views in a database can be identified using the SHOW FULL TABLES statement or by querying the information_schema.views table.

Using SHOW FULL TABLES

```
USE database_name;

SHOW FULL TABLES

WHERE Table_type = 'VIEW';
```

The SHOW FULL TABLES statement lists the tables and views in the selected database. The WHERE condition restricts the result to objects whose type is VIEW.

Using INFORMATION_SCHEMA

Views can also be listed by querying the information_schema.VIEWS system view:

```
SELECT TABLE_NAME

FROM information_schema.VIEWS

WHERE TABLE_SCHEMA = 'database_name';
```

The above query returns the names of views belonging to the specified database.

Viewing View Definitions

The definition of each view can also be retrieved using:

```
SELECT TABLE_SCHEMA, TABLE_NAME, VIEW_DEFINITION

FROM information_schema.VIEWS

WHERE TABLE_SCHEMA = 'database_name';
```

This query returns the database/schema name, view name, and available SQL definition associated with each view.

2.4.3.2 Deleting a View

An existing view can be permanently removed from the database using the DROP VIEW statement.

Syntax:

```
DROP VIEW view_name;
```

SGOU

Example:

DROP VIEW MarksView;

The above statement permanently removes the MarksView view definition from the database. Dropping a view does not delete the data stored in its underlying tables.The base tables and their data remain unchanged.

2.4.3.3 Updating Data Through a View and Modifying Its Definition

There are two different operations that are sometimes referred to as updating a view:

1. Updating data through an updatable view
2. Modifying the definition of the view

These operations should be distinguished clearly.

1. Updating Data Through a View

If a view is updatable, its underlying data can be modified using an UPDATE statement.

UPDATE view_name

SET column1 = value1,

column2 = value2

WHERE condition;

Example

Suppose StudentView is an updatable view containing S_ID, Name, and Age.

UPDATE StudentView

SET Age = 22

WHERE S_ID = 101;

This statement modifies the data through the view. If the view is updatable, the corresponding value in the underlying base table is also modified.

2. Modifying the Definition of a View

If the purpose is to change the **query definition** of an existing view rather than modify its data, CREATE OR REPLACE VIEW can be used in DBMSs that support this syntax.

CREATE OR REPLACE VIEW view_name AS

SELECT column1, column2, ...

SGOU

FROM table_name

WHERE condition;

Example

Suppose the existing view displays only Name and Address. Its definition can be replaced to include Age:

CREATE OR REPLACE VIEW StudentView AS

SELECT Name, Address, Age

FROM StudentDetails;

The definition of StudentView is replaced with the new query, so the view now provides Name, Address, and Age.

Note: Not every SQL view is updatable. The ability to insert, update, or delete data through a view depends on the view definition and the capabilities of the particular DBMS. Views involving complex queries, aggregation, grouping, or certain joins may be read-only.

2.4.4 Rules for Updating Views in SQL

A view is updatable when the DBMS can translate an `INSERT`, `UPDATE`, or `DELETE` operation performed on the view into an unambiguous modification of the underlying table or tables. The exact rules vary among DBMSs.

The following conditions generally apply to simple updatable views:

- The view should generally be based on a single table. Views involving multiple tables may not be directly updatable, although some DBMSs support updates to certain join views.
- The SELECT statement should not contain DISTINCT, as it can make the relationship between view rows and underlying rows ambiguous.
- The view should generally not use GROUP BY or aggregate functions such as SUM(), AVG(), COUNT(), MIN(), or MAX() because the resulting rows do not directly correspond to individual rows in the underlying table.
- The view should not contain query features that prevent the DBMS from determining how a modification should be applied to the underlying table.
- For inserting records, required NOT NULL columns of the underlying table that have no default value generally need to be included in the view or otherwise supplied by the DBMS.
- The view should provide a sufficiently direct relationship between its rows and the underlying table rows so that the DBMS can determine which base-table records should be modified.

SGOU

2.4.5 Advanced Techniques with Views

Views can also be used to perform data modification operations when they satisfy the required conditions for updatability. Depending on the view definition and the DBMS, UPDATE, INSERT, and DELETE operations may be performed through an updatable view.

2.4.5.1 Updating Data Through Views

An updatable view can be used to modify data in its underlying table. For example, suppose MarksView contains the NAME and ADDRESS fields.

The address of a particular student can be changed through the view using the following statement:

```
UPDATE MarksView

SET ADDRESS = 'Perth'

WHERE NAME = 'Liam';
```

The above statement changes the address associated with the student named Liam. If the view is updatable, the corresponding data in the underlying base table is also modified.

The modified information can then be viewed using:

```
SELECT * FROM MarksView;
```

The result displays the updated information available through the view.

Note: In practice, it is preferable to use a unique identifier such as S_ID rather than NAME in the WHERE clause, because multiple students may have the same name.

2.4.5.2 Inserting Data into Views

A row can be inserted through an updatable view using the INSERT INTO statement, provided that the view and the underlying table satisfy the required conditions.

Example:

```
INSERT INTO MarkView(NAME, ADDRESS)

VALUES("John","German");
```

If the view is updatable and all required conditions are satisfied, the corresponding row is also inserted into the underlying table.

The contents of the view can then be retrieved using:

```
SELECT * FROM MarkView;
```

The newly inserted record will be displayed along with the other records available through the view.

SGOU

2.4.5.3 Deleting a Row from a View

A row can also be deleted through an updatable view using the DELETE statement. When the operation is permitted, deleting a row through the view removes the corresponding record from the underlying base table.

For example, the previously inserted record for John can be removed using:

DELETE FROM MarkView

WHERE NAME="John";

The above statement identifies the record whose NAME is John and removes it through the view.

The remaining records can be displayed using:

SELECT * FROM MarkView;

The deleted record will no longer appear in the result.

2.4.5.4 WITH CHECK OPTION Clause

The WITH CHECK OPTION clause is used with an updatable view to ensure that INSERTand UPDATE operation performed through the view continues to satisfy the condition specified in the view definition.

If an operation attempts to create or modify a row so that it no longer satisfiesthe view's WHERE condition, the operation is rejected.

Example:

CREATE VIEW SampleView AS

SELECT S_ID, NAME

FROM StudentDetails

WHERE NAME IS NOT NULL

WITH CHECK OPTION;

The above statement creates SampleView containing only records for which NAME is not NULL.

An update can then be performed through the view:

UPDATE SampleView

SET NAME = 'Mark'

WHERE S_ID = 2;

The update is permitted because the new value 'Mark' satisfies the condition NAME IS

SGOU

NOT NULL.

For example, an operation such as:

UPDATE SampleView

SET NAME = NULL

WHERE S_ID = 2;

would be rejected because the resulting row would no longer satisfy the condition defined in the view.

Thus, the WITH CHECK OPTION clause helps ensure that modifications made through an updatable view remain consistent with the condition specified in the view definition.

2.4.6 TCL (Transaction Control Language) commands and views

In SQL, TCL (Transaction Control Language) commands are used to manage transactions and control changes made to the database. TCL commands are not specific to views, but they can affect transactions involving data modification operations performed through updatable views.

The main TCL commands are:

TCL Command	Purpose
COMMIT	Permanently saves the changes made during the current transaction.
ROLLBACK	Cancels uncommitted changes made during the current transaction.
SAVEPOINT	Creates an intermediate point within a transaction to which the transaction can be rolled back.
ROLLBACK TO SAVEPOINT	Reverts the transaction to a previously created savepoint without cancelling the entire transaction.
SET TRANSACTION	Specifies transaction properties such as isolation level or access mode.

Example using a view

Suppose the following view has been created:

CREATE VIEW Employee_View AS

SELECT Employee_ID, Employee_Name, Salary

FROM Employee;

If the view is updatable, modifications made through the view can participate in a transaction.

SGOU

Using COMMIT

START TRANSACTION;

UPDATE Employee_View

SET Salary = Salary + 5000

WHERE Employee_ID = 101;

COMMIT;

Here, COMMIT makes the changes performed during the transaction permanent, subject to the DBMS transaction rules.

Using ROLLBACK

If the changes need to be cancelled:

START TRANSACTION;

UPDATE Employee_View

SET Salary = Salary + 5000

WHERE Employee_ID = 101;

ROLLBACK;

The update is discarded, provided the operation is transactional and the DBMS and underlying storage system support rollback.

Using SAVEPOINT with a view

START TRANSACTION;

UPDATE Employee_View

SET Salary = Salary + 5000

WHERE Employee_ID = 101;

SAVEPOINT S1;

UPDATE Employee_View

SET Salary = Salary + 3000

WHERE Employee_ID = 102;

ROLLBACK TO SAVEPOINT S1;

COMMIT;

SGOU

In this example, the first update remains part of the current transaction, while the second update is undone when the transaction is rolled back to S1. The first update becomes permanent when COMMIT is executed.

Important Distinction

CREATE VIEW, ALTER VIEW, and DROP VIEW are DDL commands, not TCL commands. DDL commands are used to define or modify database objects, whereas TCL commands are used to control transactions.

Summarised Overview

A view is a logical subset of data obtained from one or more database tables and presented as a virtual table. Unlike a physical table, a view generally does not store a separate copy of the data but retrieves information using its defined SQL query. To users, a view appears similar to a regular table with rows and columns. Views are useful for restricting access to selected rows or columns and improving database security. They can simplify complex queries by combining information from multiple tables into a single logical representation. Views also provide structural simplicity, consistency, data integrity, and a degree of logical data independence. However, complex views may require additional processing and may have restrictions on data modification. A view can be created using the CREATE VIEW statement and may be based on a single table or multiple tables. SQL provides commands such as SHOW FULL TABLES, information_schema.views, DROP VIEW, and CREATE OR REPLACE VIEW for managing views. An updatable view may allow users to perform UPDATE, INSERT, and DELETE operations, depending on how the view is defined. Certain conditions, such as avoiding GROUP BY, ORDER BY, DISTINCT, and complex queries, generally need to be satisfied for a view to be directly updatable. The WITH CHECK OPTION clause ensures that modifications made through an updatable view continue to satisfy the condition specified in its definition. TCL commands such as COMMIT, ROLLBACK, and SAVEPOINT can control transactions involving changes made through updatable views, while commands such as CREATE VIEW, ALTER VIEW, and DROP VIEW are DDL commands used to manage view definitions.

SGOU

Assignments

1. Explain the concept of a SQL view and describe how a view differs from a physical database table. Illustrate your answer with a suitable example.

2. Discuss the major advantages of using views in SQL. Explain how views provide security, query simplicity, structural simplicity, consistency, data integrity, and logical data independence.

3. Explain the disadvantages of views with suitable examples. Discuss how complex view definitions can affect performance and why some views may have restrictions on data modification.

4. Describe the procedure for creating a view using the CREATE VIEW statement. Explain the syntax and demonstrate how to create and retrieve a view using a single database table.

5. Explain how a view can be created using multiple tables. Using suitable SQL statements, describe how information from StudentDetails and StudentMarks can be combined to create and retrieve a view.

6. Explain the different operations used for managing views in SQL. Describe how views can be listed, examined, deleted using DROP VIEW, and modified using CREATE OR REPLACE VIEW.

7. Discuss the concept of updatable views in SQL. Explain the conditions that generally need to be satisfied for a view to support UPDATE, INSERT, and DELETE operations, with suitable examples.

8. Explain how data can be modified through an updatable view. Demonstrate the use of UPDATE, INSERT INTO, and DELETE statements with suitable examples and explain how these operations affect the underlying table.

9. Explain the purpose and working of the WITH CHECK OPTION clause in SQL views. Using a suitable example, describe how it helps ensure that INSERT and UPDATE operations satisfy the condition specified in the view definition.

10. Discuss the use of TCL commands with updatable views. Explain the roles of COMMIT, ROLLBACK, SAVEPOINT, ROLLBACK TO SAVEPOINT, and SET TRANSACTION, and distinguish TCL operations from view related DDL commands such as CREATE VIEW, ALTER VIEW, and DROP VIEW.

SGOU

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, 7th edition.

Suggested Reading

1. Concept of Database Management, Pratt, Thomson Learning, 5th Edition.

2. Database System Concepts - Silberchatz, Korth and Sudarsan, Fifth Edition, McGraw Hill, 2006

3. The Complete Reference SQL - James R Groff and Paul N Weinberg

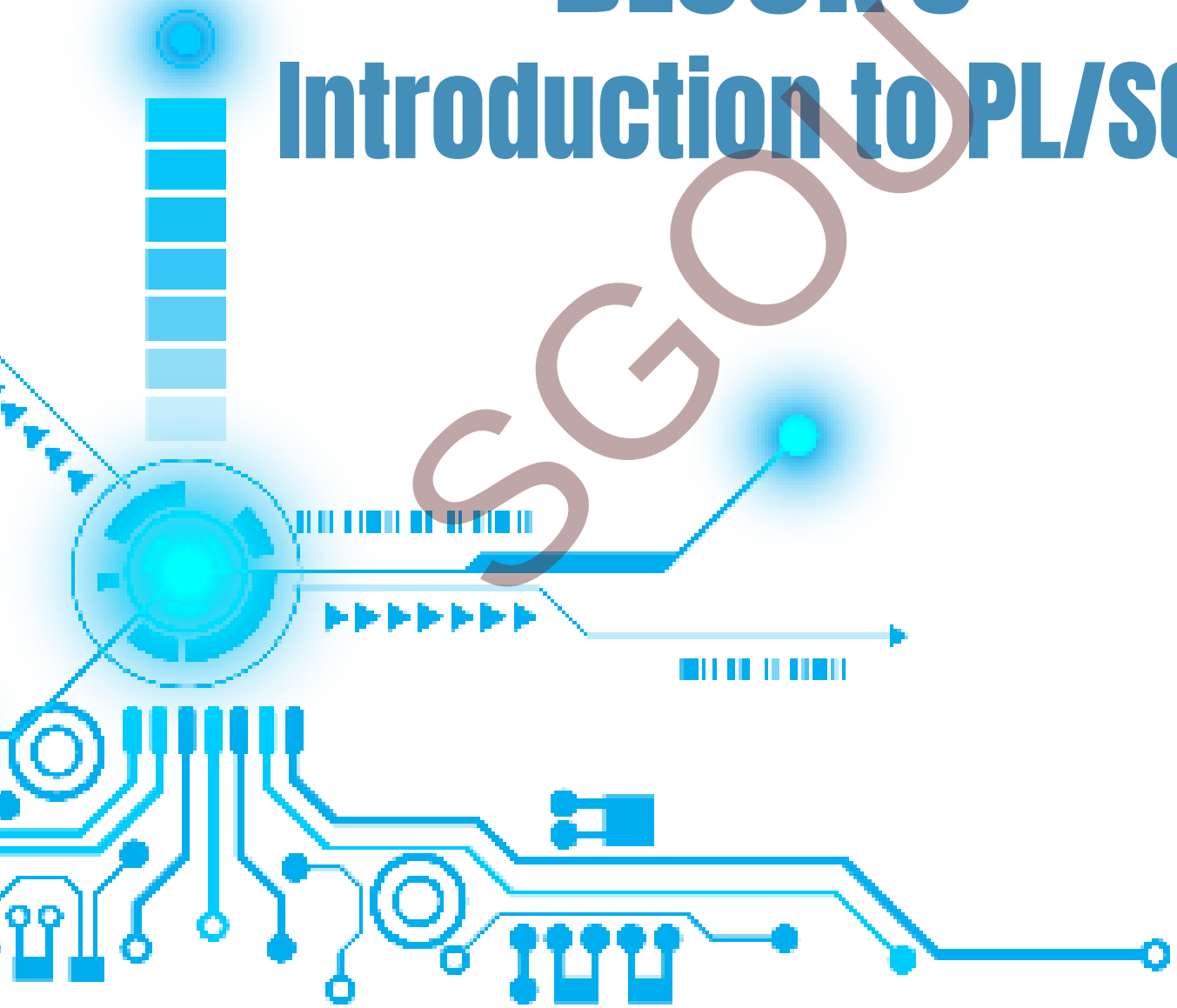
SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

BLOCK 3
Introduction to PL/SQL



SGOU

1 UNIT

Introduction to PL/SQL

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the basic concepts, features, and advantages of PL/SQL
- describe the structure of a PL/SQL block and explain the purpose of its different sections
- use variables and appropriate data types in PL/SQL programs
- apply control structures such as IF, CASE, and loops to develop PL/SQL programs

Background

SQL is mainly used to retrieve, insert, update, and delete data in a database. However, database applications often require additional programming capabilities such as decision-making, repetition, calculations, variable handling, and error handling. SQL alone does not provide the full range of procedural programming features required for many database applications. PL/SQL addresses this requirement by combining SQL with procedural programming features.

PL/SQL, which stands for Procedural Language/Structured Query Language, is Database's procedural extension to SQL. It is used to develop programs that interact with and process data in an Oracle database.PL/SQLprovides programming features such as variables, conditional statements, loops, and exception handling. PL/SQL programs are organized into blocks and can perform SQL operations together with procedural logic.

SGOU

Learning PL/SQL helps learners understand how database operations can be combined with programming logic. It supports decision-making, repeated execution of statements, calculations, error handling, and code reuse.PL/SQL can also reduce communication between an application and the database by allowing multiple SQL and procedural statements to be executed as a block on the database server

Keywords

Oracle Database, DECLARE, BEGIN, EXCEPTION, END, Variable, Datatype, NUMBER, VARCHAR2, CHAR, DATE, BOOLEAN

Discussion

SGOU

3.1.1 Basics of PL/SQL

Imagine that your college stores all student information in an Oracle database. The Head of the Department wants to increase the internal marks of students who scored above 80 and display a message such as "Marks Updated Successfully."

Can SQL alone perform both tasks?

SQL can be used to update the marks, but SQL by itself does not provide the procedural programming features needed for decision-making, repeated processing, variable handling, and program-controlled output.

For such database programming tasks, Oracle provides PL/SQL. PL/SQL combines the data manipulation capabilities of SQL with procedural programming features such as variables, conditions, loops, and exception handling.

What is PL/SQL?

PL/SQL stands for Procedural Language/Structured Query Language. It is OracleDatabase's procedural extension to SQL that adds programming features such as variables, conditional statements, loops, and exception handling.

Using PL/SQL, programmers can write database programs that combine SQL statements with procedural programming logic.This allows database operations to be performed together with decision-making, repeated processing, calculations, and error handling.

Think of SQL as a language mainly used to ask questions and perform operations on database data.

PL/SQL is like a smart assistant that can:

- Ask the database for information.
- Make decisions based on the information.
- Perform calculations.
- Repeat tasks using loops.
- Handle errors.
- Display messages.

Example

Suppose we want to display a welcome message using PL/SQL.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL');
END;
/
```

Output

Welcome to PL/SQL

3.1.2 Advantages of PL/SQL

Every programming language is designed to solve problems efficiently. PL/SQL provides many advantages over using SQL alone because it combines database operations with programming capabilities.

1. **Reduced Network Communication**

A PL/SQL block can contain multiple SQL and procedural statements and can be sent to the database server as a unit. This reduces repeated communication between the application and the database and can improve performance, particularly when several related operations are performed.

Example: Updating the salaries of 100 employees using a single PL/SQL program instead of sending 100 separate requests from the application to the database.

SGOU

2. **Supports Decision Making And Repetition**

PL/SQL supports conditional processing using statements such as IF and CASE, as well as repeated processing using loops.

Example: If a student's mark is greater than 50, display PASS; otherwise, display FAIL.

3. **Error Handling**

PL/SQL provides exception-handling mechanisms that allow runtime errors to be handled in a controlled manner.

Example: If a program attempts to divide a number by zero, an exception handler can detect the error and display an appropriate message.

4. **Code Reusability**

PL/SQL code can be organized into reusable procedures, functions, packages, and triggers. This reduces code duplication and makes database programs easier to maintain.

Example: A salary-calculation procedure can be created once and reused whenever salary calculations are required.

5. **Improved Security**

PL/SQL code can be organized into reusable procedures, functions, packages, and triggers. This reduces code duplication and makes database programs easier to maintain.

Example: A salary-calculation procedure can be created once and reused whenever salary calculations are required.

3.1.3 Structure of a PL/SQL Block

A PL/SQL program is organized into one or more blocks. A block organizes the program into different sections, making it easier to understand and maintain.

Think of preparing a meal:

- First, collect the ingredients.
- Then, cook the food.
- If something goes wrong, fix it.
- Finally, finish the cooking.

A PL/SQL block works in the same way.

SGOU

General Structure of a PL/SQL Block

```
DECLARE
  -- Variables

BEGIN
  -- Executable statements

EXCEPTION
  -- Error handling

END;
/
```

3.1.3.1 Sections of a PL/SQL Block

A PL/SQL program is organized into one or more blocks. A block is the basic unit of a PL/SQL program and consists of different sections, each serving a specific purpose. Understanding these sections helps beginners write well-organized and maintainable PL/SQL programs.A PL/SQL block has three main sections:

1. Declaration section (DECLARE) – Optional
2. Executable section (BEGIN ... END) – Mandatory
3. Exception-handling section (EXCEPTION) – Optional

The general structure is:

1. DECLARE Section

The DECLARE section is the first part of a PL/SQL block. It is used to declare variables, constants, cursors, and user-defined datatypes before they are used in the block.

Think of this section as preparing the ingredients before cooking. Just as a cook gathers vegetables, spices, and utensils before starting to cook, a programmer declares the required variables and other items before writing the program logic.

SGOU

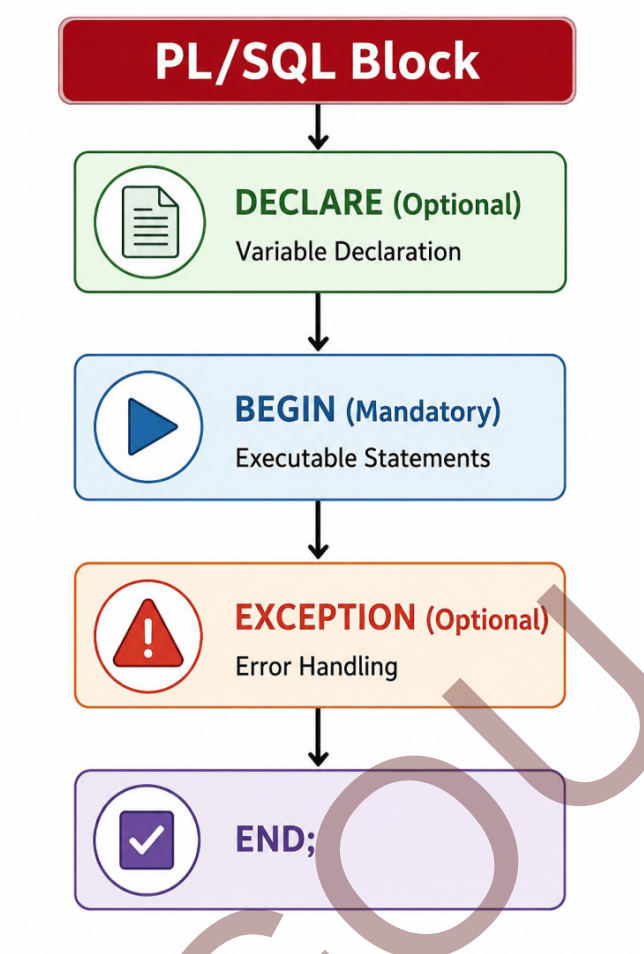


Fig 3.1.1 PL/SQL Block

Purpose of DECLARE Section

- Declares variables.
- Declares constants.
- Declares cursors.
- Defines user-defined datatypes.

If the block does not require any declarations, the DECLARE section can be omitted.

- student_name is declared as a VARCHAR2 variable to store text.
- marks is declared as a NUMBER variable to store numeric values.
- These variables are declared in the DECLARE section and used in the BEGIN section.

Imagine filling out a student admission form. Before entering details, the form contains fields such as Name, Age, and Course. Similarly, in PL/SQL, variables are declared first so they are ready to store values.

SGOU

Syntax

DECLARE

variable_name datatype;

Example :Declaring Variables

```
DECLARE
student_name VARCHAR2(30);
  marks NUMBER;
BEGIN
student_name := 'Anu';
  marks := 85;

  DBMS_OUTPUT.PUT_LINE(student_name);
  DBMS_OUTPUT.PUT_LINE(marks);
END;
/
```

Output

Anu

85

SGOU

2. BEGIN Section

The BEGIN section contains the executable statements of a PL/SQL block. This is where the program performs calculations, assigns values to variables,executes SQL statements, displays output, and calls procedures or functions.

Think of the BEGIN section as cooking the meal after all the ingredients are ready.

Purpose of BEGIN Section

- Assign values to variables.
- Executes SQL statements.
- Performs calculations.

Syntax

```
BEGIN
executable_statements;
END;
```

Example : Executing Statements

```
DECLARE
  salary NUMBER;
BEGIN
  salary := 35000;

  DBMS_OUTPUT.PUT_LINE('Salary = ' || salary);
END;
/
```

Output

```
Salary = 35000
```

- Displays output.
- Call procedures and functions.

Every PL/SQL block must contain a BEGIN and an executable section, followed by END.

- A variable named salary is declared.
- The value 35000 is assigned in the BEGIN section.
- The value of salary is displayed using DBMS_OUTPUT.PUT_LINE().

Imagine a washing machine. After loading clothes and detergent, you press the Start button. The washing process begins. Similarly, when the BEGIN section starts, the PL/SQL program begins executing its instructions.

3. EXCEPTION Section

The EXCEPTION section is an optional section used to handle exceptions raised during the execution of a PL/SQL block. It allows errors to be handled in a controlled manner.

SGOU

Think of this section as an emergency plan. If something unexpected happens, the program can respond according to the specified exception handler.

Purpose of EXCEPTION Section

- Handles runtime exceptions.
- Allows errors to be handled in a controlled manner.
- Displays meaningful error messages.
- Improves the reliability of database programs.

If no explicit exception handling is required, the EXCEPTION section can be omitted.

Syntax

```
EXCEPTION
WHEN exception_name THEN
 statements;
```

Example : Handling Division by Zero

```
DECLARE
   num1 NUMBER := 20;
   num2 NUMBER := 0;
   result NUMBER;
BEGIN
   result := num1 / num2;
   DBMS_OUTPUT.PUT_LINE(result);
EXCEPTION
WHEN ZERO_DIVIDE THEN
 DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed.');
END;
/
```

Output

```
Error: Division by zero is not allowed.
```

SGOU

- The program attempts to divide 20 by 0.
- Oracle raises the ZERO_DIVIDE exception.
- The corresponding exception handler executes.
- A meaningful message is displayed instead of allowing the exception to remain unhandled.

When driving a car, you wear a seatbelt even though you hope never to need it. Similarly, the EXCEPTION section provides a mechanism for handling unexpected situations when an error occurs.

4. END Keyword

The END keyword marks the end of the PL/SQL block. It is not normally considered a separate section of the block.

Purpose of END Keyword

- Indicates the end of the PL/SQL block.
- Completes the block structure.
- Is required in every PL/SQL block.

Syntax

```
END;
/
```

Example

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Program Completed Successfully');
END;
/
```

Output

```
Program Completed Successfully
```

Here, END; marks the end of the PL/SQL block.

The / is not part of the PL/SQL language itself. In tools such as SQL*Plus and SQL Developer script mode, it is used to submit the completed PL/SQL block for execution.

Real-Life Example: When writing an examination, you finish your answer with a final sentence or a full stop. Similarly, every PL/SQL block must be completed with the END keyword.

SGOU

Note: END; terminates the PL/SQL block, while / is used by tools such as SQL*Plus and SQL Developer script mode to submit the completed block for execution.

3.1.4 Variables and Data Types

Imagine you are filling an online admission form. The form asks you to enter your Name, Age, Date of Birth, and Course.

Can all these values be stored in one box?

No.

The student's name is text, age is a number, and date of birth is a date. Therefore, each piece of information requires a separate storage location.

Similarly, while writing a PL/SQL program, different types of information require separate storage locations. These storage locations are called variables.

3.1.4.1 Variables

A variable is a named memory location used to store data temporarily during the execution of a PL/SQL program. The value stored in a variable can change during program execution, which is why it is called a variable.

Whenever a PL/SQL program performs calculations, stores input values, or displays information, variables are used to hold the required data. Variables make programs flexible because the same program can work with different values without changing the program logic.

Why Do We Need Variables?

Suppose a college wants to calculate the average marks of 500 students.

Without variables, every student's marks would need to be directly written into the program. This would make the program lengthy and difficult to modify, and difficult to maintain.

Using variables, the program can store values, perform calculations, and process data efficiently. Thus, variables make programs easier to understand, modify, and reuse.

Uses of Variables

Variables are used to:

- Store input values.
- Store calculation results.
- Perform arithmetic operations.
- Display output.
- Reuse values whenever required.

SGOU

Think of a water bottle.

A variable is like a bottle that temporarily stores water, where the bottle represents the variable and the water represents the data; similarly, a variable temporarily stores a value and can later store another value during program execution.

Declaring Variables

Before using a variable in a PL/SQL program, it must be declared in the DECLARE section.

- student_name is declared as a VARCHAR2 variable to store text.
- marks is declared as a NUMBER variable to store numeric values.
- Values are assigned to variables in the BEGIN section.
- DBMS_OUTPUT.PUT_LINE() is used to display the values.

Syntax

```
DECLARE
    variable_name data_type;
```

Example

```
DECLARE
    student_name VARCHAR2(30);
    marks NUMBER;
BEGIN
     student_name := 'Rahul';
     marks := 85;
      DBMS_OUTPUT.PUT_LINE(student_name);
      DBMS_OUTPUT.PUT_LINE(marks);
END;
/
```

Output

```
Rahul
85
```

SGOU

Rules for Naming Variables

A variable name in PL/SQL should follow certain rules to ensure that the program is easy to understand and execute correctly.

1. A variable name must begin with a letter.
2. It can contain letters, numbers, and underscores (_).
3. Spaces are not allowed in variable names.
4. SQL reserved words cannot be used as variable names.
5. Variable names should be meaningful and descriptive.

Table 3.1.1 Example for Valid and Invalid Variable Names

Valid	Invalid
student_name	1student
total_marks	student-name
age	select
salary1	first name

Assigning Values to Variables

Values are assigned to variables using the assignment operator (:=).

This statement means: Store the value 25000 in the variable salary.

Displaying Variable Values

The `DBMS_OUTPUT.PUT_LINE()` procedure is used to display the value stored in a variable.

Example

Syntax

variable_name := value;

Example:

salary := 25000;

Here, the value stored in the variable student_name is displayed on the screen.

DBMS_OUTPUT.PUT_LINE(student_name);

Output

Rahul

SGOU

3.1.4.2 Data Types

Suppose you have three containers, one for water, one for books, and one for money; just as water cannot be stored in a money purse because each container is designed for a specific purpose, variables require different data types depending on the type of information they store.

What is a Data Type?

A **d**ata type specifies the type of value that can be stored in a variable. It tells Oracle whether a variable contains numbers, text, dates, or logical values.

Data types help Oracle determine how data should be stored, validated, and processed.

For example:

- A student's name should be stored as text.
- Marks should be stored as numbers.
- Date of birth should be stored as a date value.

Why Do We Need Data types?

Data types help Oracle:

- Store data correctly.
- Save memory efficiently.
- Perform operations accurately.
- Prevent invalid data from being stored.

Different type of PL/SQL Datatypes

1. **NUMBER**

The `NUMBER` data type stores whole numbers and decimal values.

> **Syntax**
>
> marks NUMBER;

Examples: 50, 100, 98.75

Used for:

- Marks
- Salary

SGOU

- Age
- Price

2. VARCHAR2

The VARCHAR2 data type stores variable-length character strings.

Syntax

student_name VARCHAR2(30);

Examples:

Rahul, Computer Science, Kerala

Used for:

- Names
- Addresses
- Course names

3. CHAR

The CHAR data type stores fixed-length character values.

Syntax

grade CHAR(1);

Example:

A, B, C

Used for

- Grades
- Fixed-size codes

4. DATE

The DATE data type stores date and time values.

Syntax

```
joining_date DATE;
```

SGOU

Example

15-AUG-2026

Used for

- Joining date
- Date of birth
- Examination date

5. BOOLEAN

The BOOLEAN data type stores logical values.

Possible values

1. TRUE
2. FALSE
3. NULL

Used for:

- Yes/No decisions
- Pass/Fail status

Table 3.1.2 Different Types of PL/SQL Data Types

Datatype	Stores	Example
NUMBER	Numeric values	95
VARCHAR2	Variable-length text	Rahul
CHAR	Fixed-length character	A
DATE	Date and time values	15-AUG-2026
BOOLEAN	Logical value	TRUE

3.1.5 Control Structures

Imagine you are crossing a road.

If the traffic light is green, you walk.

SGOU

If it is red, you stop.

If it is yellow, you wait.

Here, your decision depends on a condition.

Similarly, a computer program also needs to make decisions and repeat certain tasks. PL/SQL uses control structures for this purpose.

What are Control Structures?

Control structures determine the order in which statements are executed in a PL/SQL program. They allow a program to make decisions, repeat tasks, and select different execution paths based on conditions.

Without control structures, statements would execute sequentially one after another without any decision-making or repetition.

Control structures are programming statements that control the flow of execution of a PL/SQL program.

Why Do We Need Control Structures?

Control structures are used when a program needs to:

- Make decisions based on conditions.
- Repeat a task multiple times.
- Select one action from different alternatives.
- Control the sequence of program execution.

Example: In a student result system:

- If the student's mark is 50 or above, display PASS.
- Otherwise, display FAIL.

This decision-making ability is achieved using control structures.

3.1.5.1 Types of Control Structures

PL/SQL mainly provides three types of control structures:

1. **Sequential Statements**

Sequential statements are the simplest type of control structure. In this structure, statements are executed one after another in the same order in which they appear in the program.

There is no decision-making or repetition. After one statement is completed, control automatically moves to the next statement.

SGOU

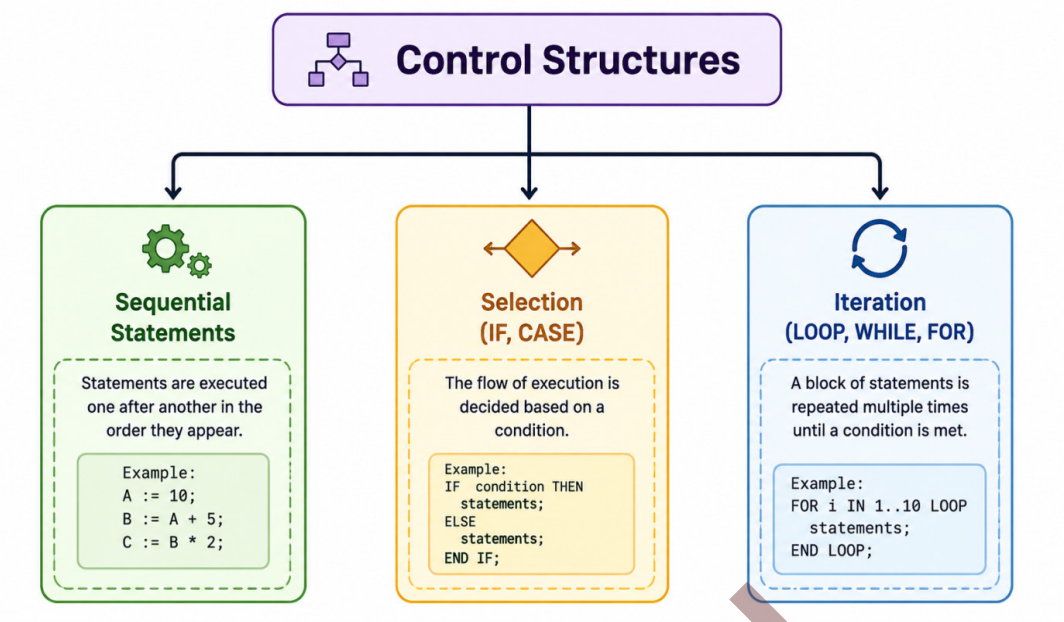


Fig 3.1.2 Different types of control structures

Example:

```
BEGIN
    A := 10;
    B := A + 5;
    C := B * 2;
END;
/
```

Execution order:

```
A := 10
  ↓
B := A + 5
  ↓
C := B * 2
```

First, A is assigned the value 10. Then B is calculated as A + 5, so B becomes 15. Finally, C is calculated as B * 2, so C becomes 30.

SGOU

In sequential execution, statements are executed from top to bottom, one after another, without any condition or repetition.

2. Selection Statements

Selection statements allow the program to choose between different actions based on conditions.

In PL/SQL, the commonly used selection statements are:

- IF -THEN
- IF-THEN-ELSE
- F-THEN-ELSIF-ELSE
- CASE

These statements allow a program to evaluate conditions and execute the appropriate set of statements.

1. IF-THEN Statement

The IF-THEN statement is used when a program needs to execute a set of statements only when a specified condition is true. If the condition is false, the statements inside the IF block are skipped.

Syntax

```
IF condition THEN
   statements;
END IF;
```

Example

```
DECLARE
  marks NUMBER := 75;
BEGIN
    IF marks >= 50 THEN
       DBMS_OUTPUT.PUT_LINE('PASS');
   END IF;
END;
/
```

SGOU

Output

PASS

The program checks whether marks are greater than or equal to 50. Since the condition is true, it prints PASS.

IF-THEN is used when an action needs to be performed only if a condition is true.

2. IF-THEN-ELSE Statement

If there are two possible outcomes, the IF-THEN-ELSE statement is used.

It evaluates a condition. If the condition is true, the statements in the THEN part are executed. If the condition is false, the statements in the ELSE part are executed.

Syntax

```
IF condition THEN
    statements_if_true;
ELSE
    statements_if_false;
END IF;
```

```
DECLARE
  marks NUMBER := 35;
BEGIN
  IF marks >= 50 THEN
    DBMS_OUTPUT.PUT_LINE('PASS');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FAIL');
  END IF;
END;
/
```

SGOU

The program checks whether marks are greater than or equal to 50. Since 35 >= 50 is false, the statements in the ELSE part are executed, and FAIL is displayed.

IF-THEN-ELSE is used when there are two possible alternatives based on a condition.

3. IF-THEN-ELSIF-ELSE Statement

The IF-THEN-ELSIF-ELSE statement is used when a program needs to evaluate multiple conditions and execute the statements associated with the first condition that is true.

It is an extension of the simple IF-THEN statement. While IF-THEN is suitable for a single condition, IF-THEN-ELSIF-ELSE is useful when there are several possible conditions and outcomes.

For example, when assigning grades to students, the program may need to check different ranges of marks:

- 90 and above → Grade A
- 75 to 89 → Grade B
- 60 to 74 → Grade C
- 40 to 59 → Grade D
- Below 40 → Fail

Such situations can be handled using an IF-THEN-ELSIF-ELSE statement.

Syntax

```
IF condition1 THEN
   statements1;
ELSIF condition2 THEN
   statements2;
ELSIF condition3 THEN
   statements3;
ELSE
   statements4;
END IF;
```

- **IF**: Specifies the first condition to be tested.
- **THEN:** Marks the beginning of statements to be executed if the condition is true.

SGOU

- **ELSIF:** Specifies another condition if the previous condition was false.
- Multiple ELSIF clauses can be used.
- **ELSE:** Executes when none of the conditions is true. It is optional.
- **END IF**: Marks the end of the IF statement.

```
DECLARE
   marks NUMBER := 82;
BEGIN
   IF marks >= 90 THEN
      DBMS_OUTPUT.PUT_LINE('Grade A');
   ELSIF marks >= 75 THEN
      DBMS_OUTPUT.PUT_LINE('Grade B');
   ELSIF marks >= 60 THEN
      DBMS_OUTPUT.PUT_LINE('Grade C');
   ELSIF marks >= 40 THEN
      DBMS_OUTPUT.PUT_LINE('Grade D');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Fail');
   END IF;
END;
/
```

Output

Grade B

In an IF-THEN-ELSIF-ELSE statement, conditions are evaluated in order, and the statements associated with the first true condition are executed.

4. CASE Statement

The CASE statement is another selection control structure in PL/SQL. It is used to select and execute one set of statements from multiple alternatives.

SGOU

A CASE statement can improve readability when multiple alternatives need to be selected. particularly when a program is comparing a single value with several possible values

For example, a program may display the name of a day based on a number:

1 → Monday

2 → Tuesday

3 → Wednesday

...

7 → Sunday

A CASE statement is well suited for this type of problem.

There are two important forms of CASE:

1. **Simple CASE statement**
2. **Searched CASE statement**

1. Simple CASE Statement

A Simple CASE statement compares one expression with different values and executes the statements associated with the matching value.

- Compare one expression with different values.
- Uses WHEN value THEN.

Syntax

```
CASE expression
   WHEN value1 THEN statements1;
   WHEN value2 THEN statements2;
   ELSE statements3;
END CASE;
```

Example

```
DECLARE
   grade CHAR(1) := 'A';
BEGIN
```

SGOU

```
    CASE grade
       WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
       WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
       WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
       ELSE DBMS_OUTPUT.PUT_LINE('Fail');
    END CASE;
END;
/
```

Output

Excellent

The value of grade is 'A', Therefore, the CASE statement matches it with WHEN 'A' and executes the corresponding statement.

A Simple CASE statement compares one expression with multiple possible values and executes the statements associated with the matching value.

3. Searched CASE Statement

A Searched CASE statement evaluates multiple conditions and executes the statements associated with the first condition that is true.

- Evaluates different conditions.
- Uses WHEN condition THEN.
- Suitable for ranges and complex conditions

Syntax

```
CASE
   WHEN condition1 THEN statements1;
   WHEN condition2 THEN statements2;
   ELSE statements3;
END CASE;
```

SGOU

Example

```
DECLARE
   marks NUMBER := 82;
BEGIN
   CASE
      WHEN marks >= 90 THEN
            DBMS_OUTPUT.PUT_LINE('Grade A');
      WHEN marks >= 75 THEN
            DBMS_OUTPUT.PUT_LINE('Grade B');
      WHEN marks >= 60 THEN
            DBMS_OUTPUT.PUT_LINE('Grade C');
      WHEN marks >= 40 THEN
            DBMS_OUTPUT.PUT_LINE('Grade D');
      ELSE
           DBMS_OUTPUT.PUT_LINE('Fail');
   END CASE;
END;
/
```

Output:

Grade B

SGOU

The program evaluates the conditions from top to bottom:

- 82 >= 90 → False
- 82 >= 75 → True

Therefore, Grade B is displayed. Once the first true condition is found, the remaining WHEN conditions are not evaluated.

A Searched CASE statement evaluates multiple conditions and executes the statements associated with the first true condition.

Table 3.1.3 Difference between simple CASE and Searched CASE

Simple CASE	Searched CASE
Compares a single expression with different values.	Evaluates multiple conditions.
Uses WHEN value THEN	Uses WHEN condition THEN
Suitable for matching fixed values.	Suitable for ranges and complex conditions
grade compared with 'A', 'B', 'C'	Example: marks >= 75

4. Iteration in PL/SQL

Iteration means repeating a group of statements multiple times according to a specified condition or iteration range.

In PL/SQL, iteration is mainly performed using loops. Loops are useful when the same operation needs to be performed repeatedly, such as displaying numbers, processing records, calculating values, or updating multiple rows.

Types of Loops in PL/SQL

There are three main types of loops:

1. Basic LOOP
2. WHILE LOOP
3. FOR LOOP

1. Basic LOOP

A Basic LOOP repeatedly executes a block of statements until an EXIT or EXIT WHEN statement terminates the loop.

Syntax

```
LOOP
   statements;
   EXIT WHEN condition;
END LOOP;
```

Example

```
DECLARE
```

SGOU

```
i NUMBER := 1;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE(i);
      i := i + 1;
      EXIT WHEN i> 5;
   END LOOP;
END;
/
```

Output

```
1
2
3
4
5
```

- 'i' starts with the value 1.
- The statements inside the loop are executed.
- The value of i is increased by 1.
- After printing 5, i becomes 6.
- The condition i > 5 becomes TRUE.
- The loop terminates.

A Basic LOOP does not have a built-in termination condition. An EXIT or EXIT WHEN statement is normally used to terminate it. If no appropriate exit statement is provided, the loop can continue indefinitely and become an infinite loop.

A Basic LOOP executes its statements repeatedly until an EXIT or EXIT WHEN statement transfers control outside the loop.

SGOU

2. WHILE LOOP

A WHILE LOOP repeatedly executes a group of statements as long as the specified condition is TRUE.

The condition is checked before each iteration. Therefore, if the condition is false initially, the statements inside the loop are not executed.

Syntax

```
WHILE condition LOOP
   statements;
END LOOP;
```

Example

```
DECLARE
   i NUMBER := 1;
BEGIN
   WHILE i<= 5 LOOP
      DBMS_OUTPUT.PUT_LINE(i);
      i := i + 1;
   END LOOP;
END;
/
```

Output

```
1
2
3
4
5
```

The condition i<= 5 is checked before each iteration.

- i = 1 → condition TRUE → execute
- i = 2 → condition TRUE → execute

SGOU

- i = 3 → condition TRUE → execute
- i = 4 → condition TRUE → execute
- i = 5 → condition TRUE → execute
- i = 6 → condition FALSE → stop

If the condition is FALSE initially, the statements inside the WHILE LOOP are not executed even once. A WHILE LOOP checks the condition before each iteration, so it may execute zero or more times.

3. FOR LOOP

A FOR LOOP is used when the number of iterations is known or can be defined using a range of values.

In a PL/SQL FOR LOOP, the loop variable is automatically declared and controlled by PL/SQL. The loop variable is assigned the values in the specified range one by one.

Syntax

```
FOR variable IN lower_value..upper_value LOOP
   statements;
END LOOP;
```

Example

```
BEGIN
   FOR i IN 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE(i);
   END LOOP;
END;
/
```

Output

```
1
2
3
4
5
```

SGOU

The loop automatically:

1. Starts i at 1.
2. Executes the statements.
3. Moves i to the next value automatically.
4. Continues through the specified range.
5. Stops after processing i = 5.

Unlike the Basic LOOP and WHILE LOOP, the programmer does not need to manually initialize or increment the loop variable.

The lower and upper bounds are included in the loop. Therefore, 1..5 executes the loop five times, with i taking the values 1, 2, 3, 4, and 5. A FOR LOOP automatically controls the loop variable and is convenient when the iteration range is known.

Table 3.1.4 Comparison of Loop Types

Loop Type	Condition Checked	Best Used When
Basic LOOP	Terminates using EXIT or EXIT WHEN	You want complete control over when the loop terminates.
WHILE LOOP	Continues while a condition is TRUE	The number of repetitions depends on a condition.
FOR LOOP	Executes over a specified range	The number/range of iterations is known

Suppose you want to print the numbers 1 to 10. The same task can be performed using the three types of loops as follows:

Basic LOOP

```
LOOP
   i := i + 1;
  EXIT WHEN i> 10;
END LOOP;
```

WHILE LOOP

```
WHILE i<= 10 LOOP
   i := i + 1;
END LOOP;
```

FOR LOOP

SGOU

```
FOR i IN 1..10 LOOP
   ...
END LOOP;
```

Main Difference

- Basic LOOP: The programmer controls the termination using EXIT or EXIT WHEN.
- WHILE LOOP: The condition is checked before each iteration.
- FOR LOOP: The range is specified directly, and the loop variable is automatically controlled.

All three loops can perform the same task, but they differ in how the repetition and termination are controlled.

Summarised Overview

PL/SQL is Oracle's Database's procedural extension to SQL that combines SQL with programming features such as variables, decision-making, loops, calculations, and exception handling. It provides several advantages, including reduced communication between the application and database server, error handling, code reusability, and support for controlled database access.. PL/SQL programs are organized into blocks consisting of an optional DECLARE section, a mandatory executable section beginning with BEGIN and ending with END, and an optional EXCEPTION section. The DECLARE section is used to define variables and other program elements, the BEGIN section contains executable statements, and the EXCEPTION handles runtime exceptions. The END keyword marks the completion of the PL/SQL block. Variables are used to store data temporarily during program execution, while data types such as NUMBER, VARCHAR2, CHAR, DATE, and BOOLEAN specify the type of data that can be stored.

PL/SQL also provides control structures to control the flow of program execution. Sequential statements execute statements in order, selection statements such as IF-THEN, IF-THEN-ELSE, IF-THEN-ELSIF-ELSE, and CASE support decision-making, and iteration statements such as Basic LOOP, WHILE LOOP, and FOR LOOP allow repeated execution of statements.

Together, these features enable programmers to develop structured reusable, and database-oriented applications using Oracle Database.

SGOU

Assignments

1. Explain PL/SQL and discuss its major advantages over SQL.
2. Explain the structure of a PL/SQL block. Describe the purpose of the DECLARE, BEGIN, EXCEPTION, and END parts with suitable examples.
3. What are variables and data types in PL/SQL? Explain the rules for naming variables and describe the NUMBER, VARCHAR2, CHAR, DATE, and BOOLEAN data types.
4. Explain the different types of control structures in PL/SQL. Discuss sequential, selection, and iteration structures with suitable examples.
5. Explain the IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF-ELSE statements in PL/SQL. Develop suitable examples to demonstrate their use.
6. Explain the Simple CASE and Searched CASE statements in PL/SQL. Compare the two forms with suitable examples.
7. What is iteration in PL/SQL? Explain the Basic LOOP, WHILE LOOP, and FOR LOOP with their syntax, working, and suitable examples.

SGOU

Reference

1. Greenwald, R., Stackowiak, R., & Stern, J. (2004). *Oracle essentials: Oracle database 10g*. O'Reilly Media.
2. Abbey, M., Corey, M. J., & Abramson, I. (2008). *Oracle database 11g: New features*. McGraw-Hill.
3. Bryla, B., & Loney, K. (2013). *Oracle Database 12c: The complete reference*. McGraw-Hill Education.
4. Price, J. (2016). *Oracle database 12c PL/SQL programming*. McGraw-Hill Education.

Suggested Reading

1. Feuerstein, S. (2014). *Oracle PL/SQL programming* (6th ed.). O'Reilly Media.

2. Urman, S. (2002). *Oracle 9i PL/SQL programming*. McGraw-Hill/Osborne.

3. Loney, K. (2014). *Oracle Database 12c: The complete reference*. McGraw-Hill Education.

4. Kyte, T., & Kuhn, D. (2015). *Expert Oracle Database architecture: Techniques and solutions for high-performance database programming* (2nd ed.). Apress.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

2
UNIT

Cursors

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the concept and fundamentals principles of cursors in PL/SQL
- differentiate between implicit and explicit cursors and describe their applications
- explain cursor attributes such as %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN
- describe REF cursors and explain their role in handling flexible result sets

SGOU

Background

When a SQL statement is executed, it may retrieve multiple rows or affect multiple rows. In many situations, a PL/SQL program needs to process these rows one at a time rather than treating them as a single group. A cursor provides a mechanism for processing a result set and keeping track of the current row during processing. For example, a teacher checking the marks of 50 students may examine each student's record individually and then move to the next student. Similarly, a cursor enables a PL/SQL program to retrieve and process records individually.

Cursors are particularly useful when working with SQL operations that involve multiple records. PL/SQL provides implicit cursors, which are automatically managed by Oracle, and explicit cursors, which are declared and controlled by the programmer. Cursor attributes provide information about the status and results of cursor processing, while REF cursors provide flexibility by allowing a cursor variable to refer to different result sets at run time.

Keywords

Cursor, PL/SQL, Implicit Cursor, Explicit Cursor, REF Cursor, Cursor Declaration, OPEN, FETCH, CLOSE, Result Set, Row-by-Row Processing

Discussion

3.2.1 Introduction to Cursors

When a SQL statement is executed, Oracle creates a private SQL area to store information required for processing the statement. A cursor provides a mechanism for accessing and managing this information and, for queries that return multiple rows, for keeping track of the current position in the result set. For example, consider a Student table containing the details of many students. If we execute a query such as: SELECT * FROM Student; The query may return many student records. In some situations, a PL/SQL program needs to examine each record separately, calculate a value, check a condition, or perform another operation. A cursor provides a way to retrieve and process the rows of a result set individually. Suppose a teacher has a list of 50 students and wants to check each student's marks. The teacher examines one student's record, checks the marks, records the result, and then moves to the next student. Similarly, a cursor allows a PL/SQL program to move through the rows of a result set and process them one at a time.

Cursors are therefore important in PL/SQL when row-by-row processing is required. Oracle automatically manages implicit cursors for SQL statements, while programmers can explicitly declare and control explicit cursors when they need more control over the processing of a multi-row query result.

A cursor is a mechanism associated with the processing of a SQL statement that allows PL/SQL to access and manage the result or processing information, particularly when rows need to be handled individually.

3.2.2 Fundamentals of Cursors

The basic idea behind a cursor is that a SQL statement may produce a set of rows, while PL/SQL may need to work with those rows individually. A cursor provides a mechanism for accessing the rows of a result set and keeping track of the current position during processing.

Consider the following query:

SGOU

```
SELECT student_id, name, marks
FROM student;
```

Suppose the query produces the following result:

Student ID	Name	Marks
101	Anu	85
102	Ravi	72
103	Meera	91

If the query returns the records of students 101, 102, and 103, the program can fetch the row for student 101 through the cursor and process it. It can then fetch the row for student 102, followed by student 103.

After the last row has been fetched, an attempt to fetch another row indicates that no more rows are available. Thus, a cursor provides a convenient mechanism for processing the rows of a result set one at a time when required.

A cursor also provides information about the status of cursor processing. For example, a PL/SQL program may need to determine whether a row was found, how many rows have been processed, or whether the cursor is currently open. PL/SQL provides cursor attributes such as %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN to obtain this information.

A cursor does not mean that only one row exists. A cursor may be associated with a result set containing multiple rows. The important idea is that the rows can be accessed and processed one at a time when required.

A cursor provides a mechanism for accessing and processing the rows of a query result, particularly when a PL/SQL program needs to handle the rows one at a time.

3.2.3 Types of Cursors

PL/SQL provides two main types of cursors:

1. **Implicit Cursor**

An implicit cursor is automatically created and managed by Oracle when a SQL statement is executed. The programmer does not need to explicitly declare, open, fetch, or close the cursor.

Implicit cursors are commonly associated with SQL statements such as INSERT, UPDATE, DELETE, and SELECT INTO.

SGOU

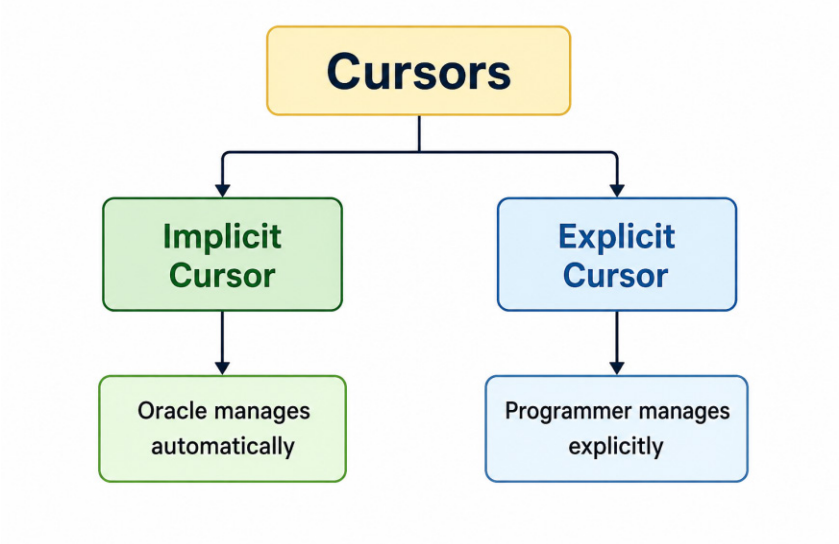


Fig 3.2.1 Types of cursor

For example:

```
UPDATE Student
SET Marks = Marks + 5
WHERE Student_ID = 101;
```

Oracle automatically manages the cursor associated with this statement.

2. Explicit Cursor

An explicit cursor is declared and controlled by the programmer. It is mainly used when a query returns multiple rows and the PL/SQL program needs to process those rows individually.

The programmer explicitly performs the following operations:

1. Declare the cursor.
2. Open the cursor.
3. Fetch rows from the cursor.
4. Close the cursor.

For example, an explicit cursor can be used to retrieve student records and process each student's information one at a time.

SGOU

3.2.3.1 Implicit Cursors

An implicit cursor is a cursor that Oracle automatically creates and manages when a SQL statement such as a DML statement or a SELECT INTO statement is executed in PL/SQL. The programmer does not need to explicitly declare, open, fetch from, or close the cursor.

Implicit cursors are commonly associated with DML statements such as INSERT, UPDATE, and DELETE. They are also used with SELECT INTO statements that retrieve a single row.

Example of an Implicit Cursor

Suppose we want to increase the marks of student 101 by 5.

```
BEGIN
   UPDATE student
   SET marks = marks + 5
   WHERE Student_ID= 101;
   DBMS_OUTPUT.PUT_LINE(
      'Rows updated: ' || SQL%ROWCOUNT
   );
END;
/
```

Implicit cursors are managed automatically by Oracle, whereas explicit cursors are declared and controlled by the programmer.

When Oracle executes the UPDATE statement, it automatically manages associated implicit cursor..

The programmer does not need to explicitly:

- Declare the cursor.
- Open the cursor.
- Fetch data from the cursor.
- Close the cursor.

Oracle performs these operations as required for the SQL statement.

SGOU

Implicit Cursor Attribute

The expression, SQL%ROWCOUNT is an implicit cursor attribute. It indicates the number of rows affected by the most recently executed SQL statement.

If one student is updated, the output will be "Rows updated: 1"

The special cursor name SQL refers to the implicit cursor associated with the most recently executed SQL statement in the current PL/SQL context.

In implicit cursor processing, Oracle automatically manages the cursor, while the programmer can use cursor attributes such as SQL%ROWCOUNT to obtain information about the statement's execution.

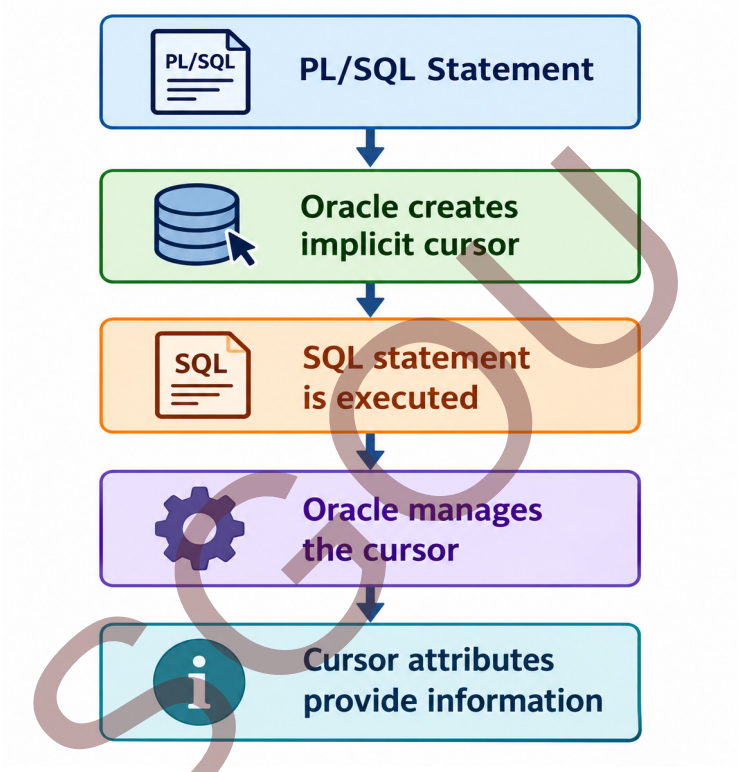


Fig 3.2.2 PL/SQL Execution process with implicit cursor

This is a good place to introduce the other implicit cursor attributes later: SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT.

3.2.3.2 Explicit Cursors

An explicit cursor is a cursor that is explicitly declared and controlled by the programmer. It is useful when a SQL query returns multiple rows and the PL/SQL program needs to process the rows individually.

For example, suppose the Student table contains 100 students and we want to display the name and marks of every student. An explicit cursor can be used to retrieve and process one student record at a time.

An explicit cursor provides greater control over the processing of a query result because the programmer determines:

- When the cursor is opened.
- When a row is fetched.
- When the cursor is closed.

The basic steps involved in using an explicit cursor are:

1. **DECLARE** – Define the cursor and its query.
2. **OPEN** – Execute the cursor query and prepare the result set.
3. **FETCH** – Retrieve rows from the result set one at a time.
4. **CLOSE** – Release the cursor after processing is complete

An explicit cursor gives the programmer control over declaring, opening, fetching, and closing a cursor for processing a query result containing multiple rows.

Steps in an Explicit Cursor

An explicit cursor generally follows four main steps:

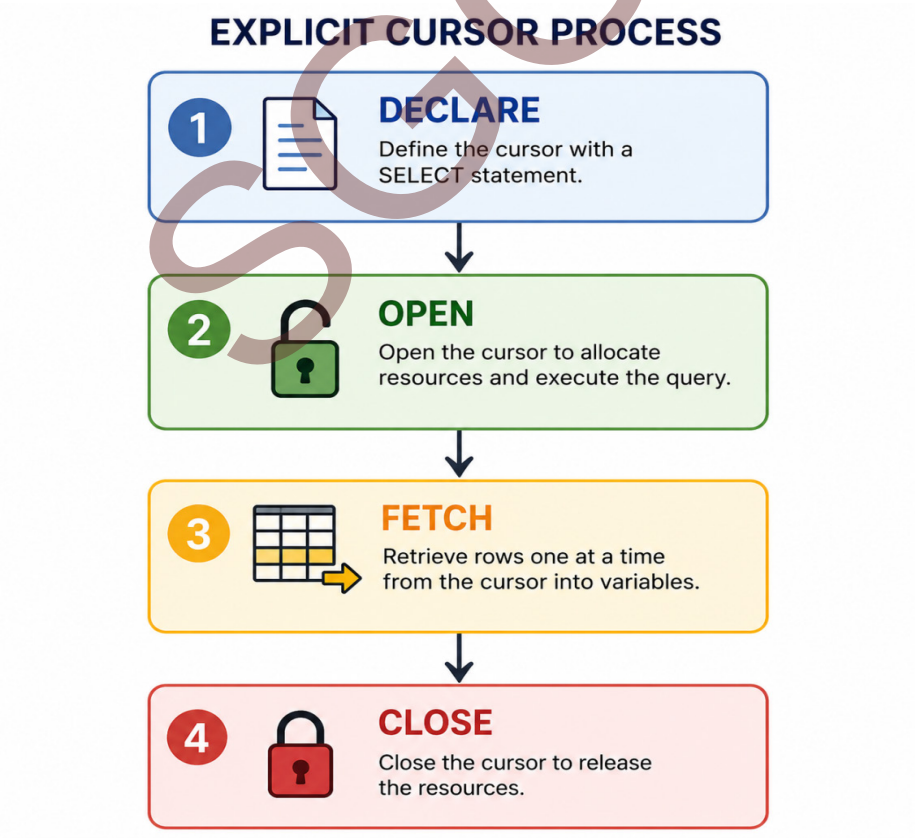


Fig 3.2.3 Steps of explicit cursor

SGOU

1. Cursor Declaration

The cursor is first declared in the DECLARE section of the PL/SQL block. The declaration gives the cursor a name and specifies the SQL query that will be used to retrieve the required data.

Syntax

```
CURSOR cursor_name IS
   SELECT column1, column2, ...
   FROM table_name
   WHERE condition;
```

Example

```
CURSOR c_student IS
   SELECT student_id, name, marks
   FROM student;
```

Here:

- c_student is the name of the cursor.
- The SELECT statement specifies the query associated with the cursor.
- student_id, name, and marks are the columns to be retrieved.
- The query is not executed when the cursor is declared. It is executed when the cursor is opened.

Cursor declaration defines the query, while the OPEN operation executes the query and makes the result set available for fetching.

2. Cursor Opening

After declaring the cursor, it is opened using the OPEN statement.

Syntax

```
OPEN cursor_name;
```

Example

```
OPEN c_student;
```

Opening the cursor causes Oracle to execute the query associated with the cursor and make the resulting rows available for processing. After the cursor is opened, rows can

SGOU

be retrieved using the FETCH statement.

The OPEN statement executes the cursor's query and prepares its result set for fetching and processing. A cursor must be open before rows can be fetched from it.

3. Fetching the Data

After opening the cursor, the FETCH statement is used to retrieve rows from the cursor's result set.

Syntax

FETCH cursor_name INTO variable1, variable2, ...;

Example

FETCH c_student INTO v_id, v_name, v_marks;

Here:

- c_student is the name of the cursor.
- v_id, v_name, and v_marks are PL/SQL variables.
- The values retrieved from the current row are stored in the corresponding variables.

The first FETCH retrieves the first available row. Each subsequent FETCH retrieves the next row in the result set.

A FETCH statement retrieves the current row from an open cursor and stores its column values in the specified PL/SQL variables. After the fetch, the cursor moves forward so that the next FETCH can retrieve the next row.

4. Closing the Cursor

After all required rows have been processed, the cursor is closed using the **CLOSE** statement.

Syntax

CLOSE cursor_name;

Example

CLOSE c_student;

Closing the cursor releases the resources associated with the cursor and its result set.

A cursor must be open before it can be fetched from, and it should be closed after the required rows have been processed.

SGOU

Suppose the student table contains the following records:

Student ID	Name	Marks
101	Anu	85
102	Ravi	72
103	Meera	91

The following program displays each student's details using an explicit cursor.:

```
DECLARE
   CURSOR c_student IS
      SELECT student_id, name, marks
      FROM student;
v_id  student.student_id%TYPE;
v_name  student.name%TYPE;
v_marks  student.marks%TYPE;
BEGIN
   OPEN c_student;
   LOOP
      FETCH c_student INTO v_id, v_name, v_marks;
      EXIT WHEN c_student%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(
           v_id || ' ' || v_name || ' ' || v_marks
      );
   END LOOP;
   CLOSE c_student;
END;
/
```

SGOU

Output

101 Anu 85

102 Ravi 72

103 Meera 91

1. Declare: c_student is declared with a SELECT statement.
2. Open: OPEN c_student executes the query and prepares the result set.
3. Fetch: FETCH retrieves one row at a time into v_id, v_name, and v_marks.
4. Check: c_student%NOTFOUND is checked after each fetch.
5. Process: If a row is available, its details are displayed.
6. Repeat: The loop continues fetching the next row.
7. Terminate: When there are no more rows, %NOTFOUND becomes TRUE, and the loop terminates.
8. Close: CLOSE c_student closes the cursor and releases its associated resources.

3.2.4 Cursor Attributes

Cursor attributes provide information about the status of a cursor and the result of its most recent operation. They are particularly useful for controlling cursor processing and determining whether rows were found or processed.

The important cursor attributes are:

Table 3.2.1 Different types of cursor attributes

Attribute	Meaning
%FOUND	Indicates whether the most recent fetch returned a row or whether the most recent SQL statement affected at least one row.
%NOTFOUND	Indicates whether the most recent fetch did not return a row or whether the most recent SQL statement affected no rows.
%ROWCOUNT	Indicates the number of rows fetched or affected by the relevant cursor operation.
%ISOPEN	Indicates whether an explicit cursor is currently open.

SGOU

1. %FOUND

The %FOUND attribute indicates whether the most recent operation was successful in finding or affecting a row. For an explicit cursor, it becomes TRUE when the most recent FETCH successfully retrieves a row.

Example

```
FETCH c_student INTO v_id, v_name, v_marks;
IF c_student%FOUND THEN
   DBMS_OUTPUT.PUT_LINE('Row fetched successfully');
END IF;
```

2. %NOTFOUND

The %NOTFOUND attribute indicates whether the most recent fetch did not return a row.

It is commonly used to terminate a loop when all rows have been processed.

Example

```
FETCH c_student INTO v_id, v_name, v_marks;
EXIT WHEN c_student%NOTFOUND;
```

This means that the loop terminates when the cursor has no more rows to fetch.

Important: In an explicit cursor, %NOTFOUND becomes TRUE when a FETCH attempts to retrieve a row after the last available row.

3. %ROWCOUNT

The %ROWCOUNT attribute indicates the number of rows fetched by an explicit cursor or affected by an implicit SQL statement.

For example:

```
DBMS_OUTPUT.PUT_LINE(c_student%ROWCOUNT);
```

If three student records have been fetched, the value of:

c_student%ROWCOUNT will be 3.

For an implicit cursor, the SQL%ROWCOUNT attribute can be used:

SGOU

```
BEGIN
   UPDATE Student
   SET Marks = Marks + 5
   WHERE Student_ID = 101;

   DBMS_OUTPUT.PUT_LINE(
      'Rows updated: ' || SQL%ROWCOUNT
   );
END;
/
```

If one row is updated, the output is: Rows updated: 1

4. %ISOPEN

The %ISOPEN attribute indicates whether an explicit cursor is currently open.

Example

```
IF c_student%ISOPEN THEN
   DBMS_OUTPUT.PUT_LINE('Cursor is open');
END IF;
```

If c_student is currently open, the message Cursor is open is displayed.

After executing: CLOSE c_student;

the expression: c_student%ISOPEN

becomes FALSE.

Cursor attributes allow a PL/SQL program to obtain information about whether rows were found, how many rows were processed, and whether an explicit cursor is open.

3.2.5 REF Cursors

A REF CURSOR is a special type of cursor variable in PL/SQL that can refer to a result set returned by a SQL query. It provides a flexible way to process query results because the query can be associated with the cursor at run time.

SGOU

In a normal explicit cursor, the SQL query is specified when the cursor is declared. In contrast, a REF CURSOR is a cursor variable whose query can be specified when the cursor is opened.

Why Use REF CURSOR?

REF CURSORs are useful when a PL/SQL program needs to work with different query results depending on the requirement.

For example, consider an application that allows the user to select the type of information to display:

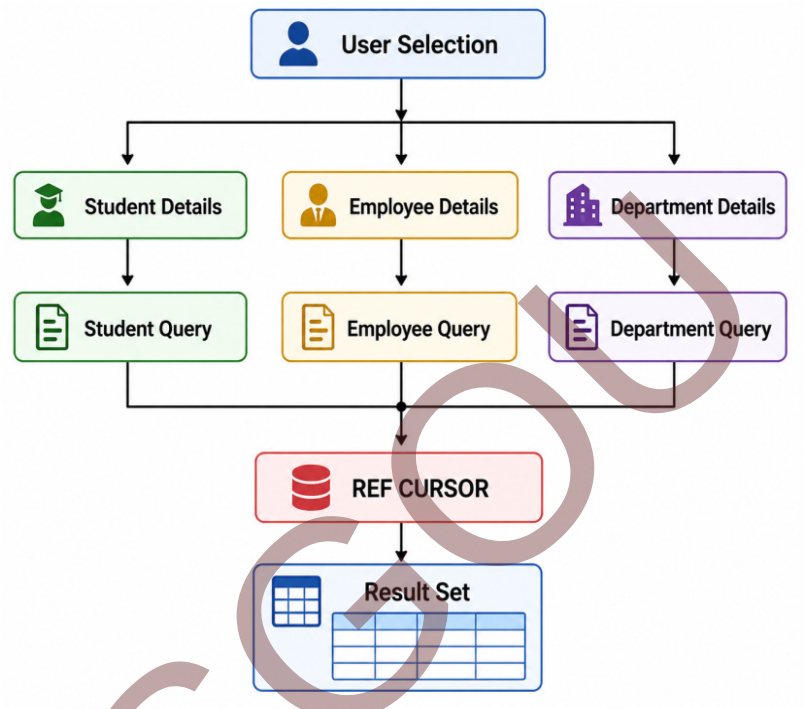


Fig 3.2.3 User selection to result set

The appropriate query can be associated with the REF CURSOR at run time.

```
DECLARE
   TYPE ref_cursor IS REF CURSOR;
   c_student ref_cursor;
   v_id    Student.student_id%TYPE;
   v_name  Student.name%TYPE;
   v_marks Student.marks%TYPE;
BEGIN
   OPEN c_student FOR
```

SGOU

```
        SELECT student_id, name, marks
        FROM student;
    LOOP
        FETCH c_student INTO v_id, v_name, v_marks;
        EXIT WHEN c_student%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(
            v_id || ' ' || v_name || ' ' || v_marks
        );
    END LOOP;
    CLOSE c_student;
END;
/
OPEN c_student FOR
    SELECT student_id, name, marks
    FROM Student;
```

Here, the query is associated with the REF CURSOR when the cursor is opened. The cursor can then be used to fetch the rows from the result set.

How It Works

1. Declare: A REF CURSOR type is defined.
2. Create a cursor variable: c_student is declared using the REF CURSOR type.
3. Open: OPEN ... FOR associates a SQL query with the cursor variable.
4. Fetch: Rows are retrieved from the result set.
5. Process: The retrieved values are processed.
6. Close: The cursor is closed after processing is complete.

An explicit cursor has its query defined when the cursor is declared, whereas a REF

SGOU

CURSOR is a cursor variable to which a query can be associated when the cursor is opened.

Table 3.2.2 Comparison of Implicit Cursor, Explicit Cursor and REF Cursor

Feature	Implicit Cursor	Explicit Cursor	REF Cursor
Created / Declared by	Oracle automatically	Programmer	Programmer
Managed / Controlled by	Oracle automatically	Programmer	Programmer
Query association	Automatically associated with the SQL statement	Defined when the cursor is declared	Associated when the cursor is opened
Main purpose	Processing individual SQL statements automatically	Row-by-row processing of a fixed query result	Handling flexible or dynamically selected result sets
Programmer Control	Low	High	High

SGOU

Summarised Overview

A cursor is a mechanism associated with the processing of a SQL statement. It is especially useful when a query returns multiple rows that need to be accessed and processed individually.

There are two main types of cursors: implicit cursors and explicit cursors. An implicit cursor is automatically created and managed by Oracle. An explicit cursor is declared and controlled by the programmer and generally follows four steps: Declare → Open → Fetch → Close.

Cursor attributes provide information about cursor processing. %FOUND indicates whether the most recent operation found or affected a row, %NOTFOUND indicates whether the most recent operation did not find or affect a row, %ROWCOUNT gives the number of rows fetched or affected, and %ISOPEN indicates whether an explicit cursor is currently open.

A REF CURSOR is a cursor variable that can refer to a result set returned by a SQL query. Unlike a normal explicit cursor, whose query is defined when the cursor is declared, a REF CURSOR can be associated with a query when the cursor is opened. This makes REF CURSORs useful when different result sets need to be handled at run time..

Assignments

1. Explain the concept and fundamentals of cursors in PL/SQL. Discuss their types, processing mechanism, and applications with suitable examples.
2. Explain implicit and explicit cursors in PL/SQL. Compare their characteristics, management, and applications with suitable examples.
3. Describe the complete life cycle of an explicit cursor from declaration to closure. Explain each stage with a suitable PL/SQL program.
4. Discuss cursor attributes in PL/SQL. Explain %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN and illustrate their significance with examples.
5. Explain REF CURSORs in detail. Discuss their characteristics, flexibility, and applications in processing different result sets.
6. Compare implicit cursors, explicit cursors, and REF cursors. Explain their characteristics, management, and appropriate applications with suitable examples.
7. Consider a student database containing student ID, name, and marks. Develop and explain a PL/SQL program that retrieves and displays each student's details using an explicit cursor.
8. Explain how explicit cursors facilitate row-by-row processing in PL/SQL. Illustrate the explanation using a suitable real-world example and PL/SQL program.
9. Discuss the importance of cursors in database programming. Explain how cursor attributes and REF cursors provide additional information, control, and flexibility in cursor processing.
10. Design a PL/SQL-based application that allows different categories of information to be retrieved through a REF CURSOR. Explain the processing flow and relevant PL/SQL statements.

SGOU

Reference

1. Feuerstein, S. (2014). *Oracle PL/SQL programming* (6th ed.). O'Reilly Media.
2. Loney, K. (2010). *Oracle Database 11g: The complete reference*. McGraw-Hill.
3. Oracle. *Database PL/SQL language reference*. Oracle Documentation.
4. Oracle. *PL/SQL user's guide and reference*. Oracle Documentation.

Suggested Reading

1. Feuerstein, S., & Pribyl, B. (2014). *Oracle PL/SQL programming* (6th ed.). O'Reilly Media.
2. Loney, K. (2010). *Oracle Database 11g: The complete reference*. McGraw-Hill.
3. Urman, S. (2002). *Oracle PL/SQL programming: A developer's workbook*. O'Reilly Media.
4. Kyte, T. (2005). *Expert Oracle database architecture*. Apress.

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

3
UNIT

Stored Procedures and Functions

Learning Outcomes

After completing this unit, the learner will be able to:

- define stored procedures and functions in PL/SQL
- explain the purpose and working of stored procedures
- describe the syntax and use of stored procedures and functions
- explain the use of parameters and the RETURN statement in PL/SQL
- differentiate between a stored procedure and a functions based on their purpose, parameters, return values, and usage

SGOU

Background

Imagine that a university database needs to perform the same task many times, such as displaying student details, updating marks, or calculating results. Would you write the same SQL and PL/SQL statements every time? It would be more convenient to write the required program once and reuse it whenever needed.

PL/SQL provides stored procedures and functions to support this type of reuse. A stored procedure is a named program unit stored in the database that performs a specific task. A function is a named program unit that performs a task and returns a value. Both procedures and functions can contain SQL statements and PL/SQL programming logic.

These reusable program units help organize database operations, reduce repeated code, and make PL/SQL applications easier to maintain.

In this unit, you will learn how stored procedures and functions are created and executed, how parameters are passed, and how functions return values. You will also learn the differences between procedures and functions and understand when each can be used.

Keywords

Stored Procedures, Parameters, Functions, Return Value, Procedure Execution

Discussion

3.3.1 Introduction to Stored Procedures

In a database application, the same set of SQL and PL/SQL statements may need to be executed repeatedly. For example, a university system may need to calculate a student's total marks, update student details, or generate a particular report many times. Writing the same statements repeatedly can make a program longer and more difficult to maintain.

PL/SQL provides stored procedures to solve this problem.

A stored procedure is a named PL/SQL program unit that is stored in the database and can be executed whenever required. It contains a group of SQL and PL/SQL statements written together to perform a specific task.

In simple terms, a stored procedure can be considered a reusable program stored inside the database.

Once created, it can be executed whenever the required task needs to be performed.

Example

Suppose a university frequently needs to display the details of a particular student. Instead of writing the same SQL statements each time, a stored procedure can be created once and executed whenever required.

```
CREATE OR REPLACE PROCEDURE DisplayStudent
IS
BEGIN
   SELECT ...
END;
/
```

SGOU

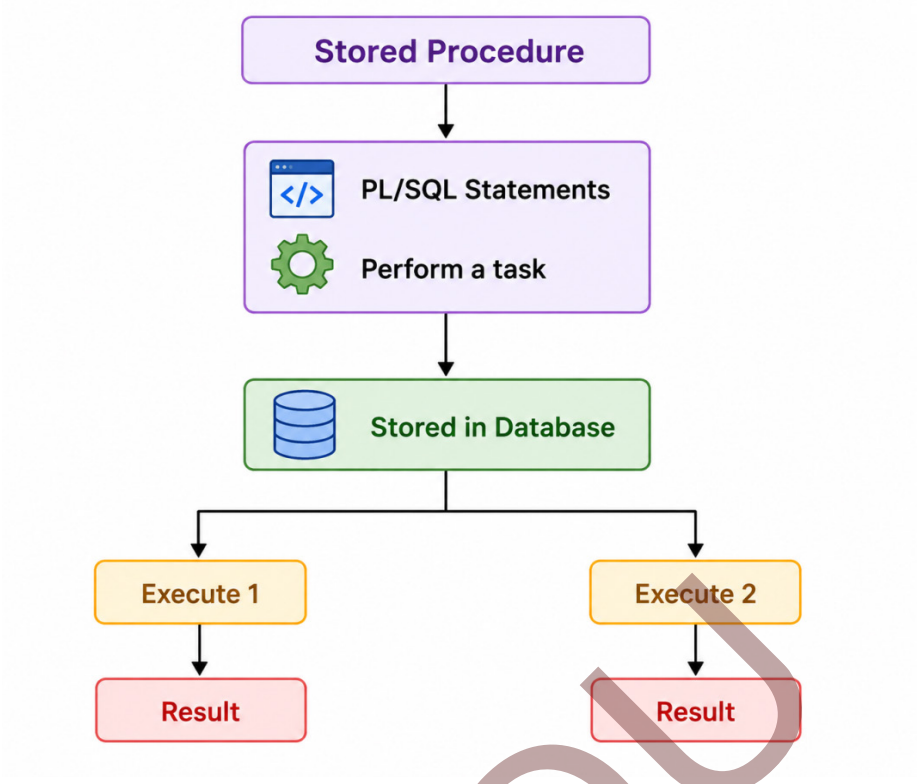


Fig. 3.3.1 Working of a Stored Procedure

How Does a Stored Procedure Work?

A stored procedure works through four simple stages:

Step 1: Define the Procedure

A stored procedure works through four simple stages:

Step 2: Store the Procedure

The procedure is created in the database using a CREATE PROCEDURE or CREATE OR REPLACE PROCEDURE statement. Once successfully created, it becomes a stored database object.

Step 3: Call the Procedure

Whenever the task needs to be performed, the stored procedure is called or executed.

Step 4: Perform the Task

The statements contained the procedure are executed and the required operation is performed.

Thus, a stored procedure helps to:

- Reuse the same set of SQL and PL/SQL statements.
- Reduce repeated code.

SGOU

- Organize database operations.
- Simplify frequently performed tasks.
- Improve maintainability of database programs.

A stored procedure is created and stored once in the database and can then be executed whenever the required task needs to be performed.

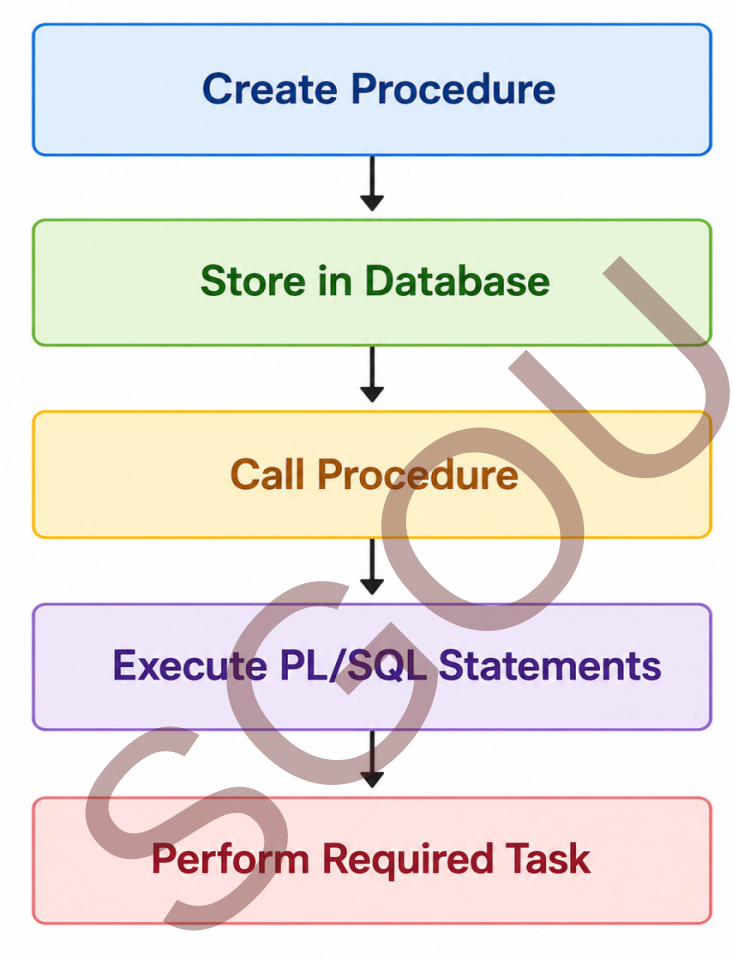


Fig 3.3.2 stages of stored procedure

3.3.2 Defining and Using Stored Procedures

A stored procedure is created in the database using the CREATE PROCEDURE statement. It is given a name and contains a set of SQL and PL/SQL statements required to perform a specific task. Once a procedure is created successfully, it is stored as a database object and can be executed whenever the required task needs to be performed.

For example, consider a university database. If a particular operation, such as displaying a message or calculating marks, updating student information, or calculating marks, needs to be performed repeatedly, the required statements can be written inside a procedure. The procedure can then be called whenever the task is required.

3.3.2.1 Defining a Stored Procedure

The general syntax for defining a stored proceedure is :

```
CREATE OR REPLACE PROCEDURE procedure_name
IS
BEGIN
    statements;
END;
/
```

Example: Creating a Simple Procedure

Suppose we want to create a procedure that displays a welcome message.

```
CREATE OR REPLACE PROCEDURE Welcome_Message
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL');
END;
/
```

Once successfully created, the procedure is stored in the database.

Executing the Procedure

The procedure can be executed using:

```
EXEC Welcome_Message;
```

or:

```
BEGIN
   Welcome_Message;
END;
/
```

Output

```
Welcome to PL/SQL
```

SGOU

In this example,

1. The procedure Welcome_Message is defined using CREATE OR REPLACE PROCEDURE.
2. The procedure is stored in the database.
3. The procedure is called using EXEC or a PL/SQL block.
4. The statements inside the procedure are executed.
5. The required output or operation is produced.

A stored procedure is created once, stored in the database, and executed whenever the required task needs to be performed.

The procedure mainly contains the following parts:

Table 3.3.1 Components of Procedure Syntax

Part	Purpose
CREATE OR REPLACE PROCEDURE	Creates a new procedure or replaces an existing procedure with the same name.
procedure_name	Specifies the name of the procedure.
IS	Marks the beginning of the procedure'x declaration section.
BEGIN	Marks the beginning of the executable section.
statements	Contains the SQL and PL/SQL statements that perform the required task.
END	Indicates the end of the procedure definition
/	Submits the complete PL/SQL block for execution in tools such as SQL*Plus and SQL Developer.

Example: Creating a Simple Stored Procedure

The IS keyword can also be replaced by **AS** in a stored procedure definition:

Syntax

```
CREATE OR REPLACE PROCEDURE procedure_name
AS
BEGIN
   statements;
END;
```

SGOU

Example

```
CREATE OR REPLACE PROCEDURE welcome_message
AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL');
END;
/
```

This procedure displays the message **"Welcome to PL/SQL"** when it is executed. The code creates a stored procedure named welcome_message. The IS keyword marks the beginning of the procedure definition, while the BEGIN keyword starts the executable section. The DBMS_OUTPUT.PUT_LINE() statement displays the message **"Welcome to PL/SQL"**. The END; statement marks the end of the procedure, and the / symbol is used to submit the complete procedure definition for execution.

3.3.2.2 Using a Stored Procedure

After a stored procedure is created, it can be executed whenever the required task needs to be performed. A procedure can be called using the EXEC command in tools such as SQL*Plus or SQL Developer.

For example, the welcome_message procedure can be executed using:

```
BEGIN
   welcome_message;
END;
/
```

Here, welcome_message is called from the executable section of the PL/SQL block.

1. The procedure is created using CREATE OR REPLACE PROCEDURE.
2. The procedure is stored in the database.
3. The procedure is called whenever the task is required.
4. The statements inside the procedure are executed.
5. The required operation or output is produced.

SGOU

Thus, a stored procedure is created and stored once in the database and can then be called repeatedly whenever required. This promotes code reuse, reduces repetition, and simplifies the execution of frequently performed tasks.

A stored procedure can be executed directly using EXEC or called from a PL/SQL block. Once stored in the database, it can be reused whenever the required task needs to be performed.

3.3.2.3 Stored Procedure with a Parameter

A stored procedure can accept values from the calling program or user. These values are passed to the procedure through parameters. Parameters make a procedure more flexible because the same procedure can be used with different input values..

For example, consider a procedure that displays the name of a student. Instead of creating a separate procedure for each student, the student name can be passed to the same procedure as a parameter.

The following procedure accepts a student name as an input parameter:

```
CREATE OR REPLACE PROCEDURE display_student
   (p_name IN VARCHAR2)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Student Name: ' || p_name);
END;
/
```

Where,

- display_student is the name of the procedure.
- p_name is the parameter used to receive the student's name.
- IN specifies that the parameter is used to pass a value into the procedure.
- VARCHAR2 specifies the data type of the parameter.
- DBMS_OUTPUT.PUT_LINE() displays the received student name.

Executing the Procedure

The procedure can be executed by passing a value to the parameter:

```
EXEC display_student('Anu');
```

SGOU

The output will be:

Student Name: Anu

The same procedure can be used with another student name:

```
EXEC display_student('Rahul');
```

The output will be:

Student Name: Rahul

Thus, a parameter allows the same stored procedure to work with different input values. This improves the reusability and flexibility of stored procedures.

3.3.2.4 Example of a Stored Procedure

Consider a student table that contains the student ID and marks. Suppose we want to display the marks of a particular student whenever required. A stored procedure can be created to perform this task.

The following procedure accepts a student ID as an input parameter and displays the marks of that student.

```
CREATE OR REPLACE PROCEDURE display_marks
   (p_id IN NUMBER)
IS
   v_marks NUMBER;
BEGIN
   SELECT marks
   INTO v_marks
   FROM Student
   WHERE student_id = p_id;
   DBMS_OUTPUT.PUT_LINE('Marks: ' || v_marks);
END;
/
```

Where,

- display_marks is the name of the procedure.

SGOU

- p_id is an IN parameter that receives the student ID.
- v_marks is a local variable used to store the marks retrieved from the table.
- The SELECT INTO statement retrieves the marks of the student whose ID matches the value of p_id.
- DBMS_OUTPUT.PUT_LINE() displays the retrieved marks.

Executing the Procedure

The procedure can be executed by passing a student ID:

```
EXEC display_marks(101);
```

If student 101 has scored 85 marks, the output will be:

Marks: 85

The same procedure can be used for another students:

For example,

```
EXEC display_marks(102);
```

If student 102 has scored 72 marks, the output will be:

Marks: 72

Thus, the same procedure can be reused for different students by passing their respective student IDs.

3.3.3 Functions

A function is a named PL/SQL program that performs a specific task and returns a value. Like a stored procedure, a function can be stored in the database and reused whenever required.

Functions are commonly used when a calculation or a specific result needs to be obtained. For example, a function can be used to calculate the total marks, average marks, salary, or percentage of a student.

A function must specify the data type of the value it returns and use the RETURN statement to return the result to the calling program.

SGOU

Simple Example

Suppose a function receives two marks, 80 and 90, calculates their total, and returns 170.

```
CREATE OR REPLACE FUNCTION calculate_total
   (mark1 IN NUMBER, mark2 IN NUMBER)
RETURN NUMBER
IS
BEGIN
   RETURN mark1 + mark2;
END;
/
```

3.3.3.1 Characteristics of Functions

A PL/SQL function has several important characteristics. The main characteristics are given below:

- **Named Program Unit:** A function is generally created to perform a particular operation or calculation.
- **Performs a Specific Task:** A function is generally created to perform a particular operation or calculation.
- **Can Accept Parameters:** A function can accept input values through parameters when required.
- **Uses the RETURN Statement:** The RETURN statement is used to return a value from the function.
- **Reusable:** Once created, a function can be called whenever the required result is needed.
- **Stored in the Database:** - A stored function is stored as a database object and can be reused by different programs or users, subject to the appropriate privileges.

3.3.3.2 Syntax to Create a Function

A function is created using the CREATE OR REPLACE FUNCTION statement. The basic syntax of a PL/SQL function is given below.

SGOU

The main parts of the function syntax are explained below.

```
CREATE OR REPLACE FUNCTION function_name

    (parameter1 datatype, parameter2 datatype, ...)

RETURN datatype

IS

BEGIN

    statements;

    RETURN value;

END;

/
```

Table 3.3.2 Components of syntax of Function

Part	Purpose
CREATE OR REPLACE FUNCTION	Creates a new function or replaces an existing function with the same name.
function_name	Specifies the name of the function.
parameter	Represents values that can be passed to the function.
RETURN datatype	Specifies the data type of the value that the function returns.
IS	Marks the beginning of the function's declaration section.
BEGIN	Marks the beginning of the executable section.
statements	Contains the SQL and PL/SQL statements required to perform the task.

SGOU

RETURN value	Returns a value from the function to the calling program.
END	Indicates the end of the function definition.
/	Submits the completed PL/SQL definition for execution in tools such as SQL*Plus and SQL Developer.

The important difference between a procedure and a function is that a function must specify a return data type and return a value using the RETURN statement.

3.3.3.3 Example of a Function

Consider a function that calculates the total marks of a student. The function accepts two marks as input and returns their total.

Example: Calculating Total Marks

```
CREATE OR REPLACE FUNCTION calculate_total
    (m1 IN NUMBER, m2 IN NUMBER)
RETURN NUMBER
IS
    total NUMBER;
BEGIN
    total := m1 + m2;
   RETURN total;
END;
/
```

Where,

- calculate_total is the name of the function.
- m1 and m2 are input parameters that receive the two marks.
- total is a local variable used to store the calculated result.

SGOU

- total := m1 + m2; adds the two marks and stores the result in total.
- RETURN total; returns the calculated value to the calling program.
- RETURN NUMBER specifies that the function returns a numeric value.

Calling the Function

The function can be called from a PL/SQL block as follows:

```
DECLARE
    result NUMBER;
BEGIN
    result := calculate_total(80, 90);
    DBMS_OUTPUT.PUT_LINE('Total Marks: ' || result);
END;
/
```

The output will be:

Total Marks: 170

Where,

- The DECLARE section declares a variable named result.
- The function calculate_total(80, 90) is called with **80** and **90** as input values.
- The function calculates 80 + 90.
- The function returns **170**.
- The returned value is stored in the variable result.
- DBMS_OUTPUT.PUT_LINE() displays the result.

SGOU

3.3.3.4 Comparison of Stored Procedure and Function

Stored procedures and functions are both named PL/SQL program units that can be stored in the database and reused whenever required. Both can contain SQL and PL/SQL statements and can accept parameters. However, they differ mainly in the way they are used and in how they handle the result of an operation.

The following table shows the major differences between a stored procedure and a function.

Table 3.3.3 Comparison of Stored procedure and Function

Feature	Stored Procedure	Function
Purpose	Mainly used to perform a specific task or operation.	Mainly used to perform a specific task and return a value..
Return Value	Does not have to return a value.	Must return a value.
RETURN Statement	Can use RETURN; to exit the procedure, but it does not return a value through the RETURN statement.	Uses the RETURN statement to return a value to the calling program
Return Data Type	Does not require a RETURN data type in its definition.	Must specify the data type of the returned value using RETURN datatype.
Parameters	Can accept parameters	Can accept parameters
Calling Method	Can be called using EXEC or from a PL/SQL block.	Can be called from a PL/SQL block, and its returned value can be assigned to a variable or used in an expression.

SGOU

Summarised Overview

Stored procedures and functions are important features of PL/SQL that help in creating reusable and organized, and maintainable database programs.. A stored procedure is a named PL/SQL program unit stored in the database that contains a group of SQL and PL/SQL statements to perform a specific task. It can be created using the CREATE OR REPLACE PROCEDURE statement and executed whenever the required task needs to be performed.

The basic structure of a procedure includes the procedure name, parameters (if required), IS or AS, the declaration section, BEGIN, executable statements, and END.

A procedure can accept parameters, allowing different input values to be passed to the same procedure. For example, a student name or student ID can be passed to a procedure to display the required information. Procedures can be executed using the EXEC command or called from a PL/SQL block.

A function is also a named PL/SQL program unit that can be stored in the database and reused whenever required.. Its main feature is that it must return a value. Functions are commonly used for calculations such as total marks, average marks, and percentage. A function is created using CREATE OR REPLACE FUNCTION and its syntax the function name, parameters (if required), RETURN datatype, IS or AS, the declaration section, BEGIN, executable statements, RETURN value, and END. The RETURN statement sends the result back to the calling program.

Both procedures and functions can accept parameters and can be reused whenever required. However, the main difference is that a procedure is primarily used to perform a task, whereas a function performs a task and returns a value.

Assignments

1. Explain the concept of stored procedures in PL/SQL. Describe how a stored procedure is created and executed with a suitable example.

2. Explain the role of parameters in stored procedures. Write and explain a PL/SQL procedure that accepts a student ID as an input parameter and displays the corresponding student marks.

3. Describe the characteristics of PL/SQL functions. Explain the purpose and use of the RETURN statement with a suitable example.

4. Write and explain a PL/SQL function to calculate the total marks or average marks of a student. Demonstrate how the function can be called from a PL/SQL block.

5. Differentiate between a stored procedure and a function based on their purpose, return value, parameters, and method of use. Illustrate the differences with suitable examples.

SGOU

Reference

1. Elmasri, Ramez, and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson Education.

2. Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill.

3. Feuerstein, Steven, and Bill Pribyl. *Oracle PL/SQL Programming*. O'Reilly Media.

Suggested Reading

1. Bayross, Ivan. *SQL, PL/SQL: The Programming Language of Oracle*. BPB Publications.

2. Narang, Rajesh. *Database Management Systems*. Prentice Hall India.

3. Oracle Database PL/SQL Language Reference

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

4
UNIT

Triggers

Learning Outcomes

After completing this unit, the learner will be able to:

- define triggers and explain their purpose in PL/SQL
- identify the main components and types of triggers
- explain DML triggers and differentiate between statement-level and row-level triggers, and between BEFORE and AFTER triggers
- describe the basic process of creating, enabling, disabling, and dropping triggers
- explain the common restrictions and limitations associated with triggers

SGOU

Background

Imagine a university database containing thousands of student records. Every day, students are registered, marks are updated, courses are added, and records are sometimes deleted. Now consider what would happen if the database administrator had to manually perform an additional action every time one of these changes occurred. For example, whenever a student's marks are changed, someone may need to record the previous marksstore the new marks, and record when the change occurred. This would be time-consuming and could easily lead to errors.

But what if the database itself could automatically detect a specified event and perform the required action??

This is exactly the idea behind a trigger. A trigger is a database program that is

automatically executed when a specified event occurs. For example, a trigger can be activated when a row is inserted, updated, or deleted from a table.

Consider a student whose marks change from 75 to 85. A trigger can automatically record the old and new marks in an audit table whenever the update occurs. Similarly, a trigger can perform a specified action when a new student is registered or an existing student record is removed.

Triggers are therefore useful for automating database actions that need to occur in response to specific events. They can be used for tasks such as maintaining audit records, enforcing certain business rules, and automatically updating related information.

In this unit, you will explore how triggers work, their main components and types, and how they are created and managed in PL/SQL. You will also learn about common restrictions and limitations associated with triggers. By the end of the unit, you will understand how triggers can be used to automate database operations and respond automatically to changes in data.

Keywords

DML Triggers, Trigger Body, Row-Level Triggers, BEFORE Trigger and AFTER Trigger, Trigger Restrictions

Discussion

3.4.1 Introduction to Triggers

In a database application, certain actions may need to be performed automatically whenever a particular event occurs. For example, in a university database, whenever the marks of a student are updated, the university may want to record the old marks and the new marks for future reference. Similarly, whenever a new student is added or an existing student record is deleted, some additional information may need to be recorded automatically.

Writing a separate program to perform these actions every time may not be convenient. PL/SQL provides triggers to perform such actions automatically in response to specified database events.

SGOU

A trigger is a special type of PL/SQL program that is stored in the database and is automatically executed , or fired, when a specified event occurs. A trigger can be associated with a table, view, or other database object, depending on the type of trigger. It is commonly used to respond automatically to events such as INSERT, UPDATE, and DELETE.

Unlike a stored procedure, a trigger is not normally executed by calling it directly. Instead, the DBMS automatically fires the trigger when its associated event occurs.

Example

Suppose the marks of a student are changed from **75 to 85**. A trigger can automatically record the old marks and the new marks in a separate audit table whenever the update occurs.

Why Are Triggers Used?

Triggers are mainly used when a particular action needs to be performed automatically in response to a specific database event.

For example, consider a university database containing a **Student** table. Whenever the marks of a student are changed, the university may want to maintain a record of the change. A trigger can automatically store the previous marks and the updated marks in an audit table.

Thus, triggers can be used to:

- Perform actions automatically in response to specified database events.
- Maintain audit records of data changes.
- Validate data before or after a specified operation, depending on the trigger.
- Enforce certain database rules.
- Maintain related information in database tables.
- Reduce repeated manual operations.

How Does a Trigger Work?

The working of a trigger can be understood through four simple stages.

Step 1: Define the Trigger

The trigger is created by specifying the event that should cause the trigger to fire and the action that should be performed.

Step 2: Create and Store the Trigger

The trigger is created and stored in the database as a database object.

SGOU

Step 3: Triggering Event Occurs

The specified event, such as INSERT, UPDATE, or DELETE, occurs on the associated table or other database object.

Step 4: Trigger Executes Automatically

When the specified event occurs, the DBMS automatically executes the statements in the trigger body.

Example

For example, when the following statement is executed:

UPDATE Student

SET marks = 85

WHERE student_id = 101;

a suitable trigger can automatically record the previous and updated marks in an audit table.

3.4.2 Components of a Trigger

A trigger contains several important components that together define when the trigger should fire, which event causes it to fire, where the event occurs, and what action should be performed. Understanding these components is important for creating and using triggers correctly.

For a typical DML trigger, the main components are:

1. Trigger Name
2. Timing
3. Triggering Event
4. Table Name
5. Trigger Level
6. Trigger Body

1. Trigger Name

A trigger must have a name by which it can be identified in the database. The name should clearly indicate the purpose of the trigger.

For example:

student_marks_audit

SGOU

Here, student_marks_audit is the name of the trigger. It indicates that the trigger is related to recording changes made to student marks.

2. Timing

Timing specifies when the trigger should fire in relation to the triggering event. The commonly used timings for DML triggers are BEFORE and AFTER.

A BEFORE trigger fires before the specified database operation is performed. An AFTER trigger fires after the specified database operation has been performed.

For example:

```
BEFORE UPDATE
```

means that the trigger fires before the update operation is performed. Similarly,

```
AFTER UPDATE
```

means that the trigger fires after the update operation has been performed..

3. Triggering Event

The triggering event specifies the database operation that causes the trigger to fire. For DML triggers, the commonly used events are:

- **INSERT** – Adds a new record to a table.
- **UPDATE** – Modifies existing data in a table.
- **DELETE** – Removes existing data from a table.

A trigger can be created to fire automatically when one or more of these operations occurs..

For example:

AFTER INSERT ON student

means that the trigger fires after a new record is inserted into the Student table.

Similarly,

AFTER UPDATE ON student

means that the trigger fires after an existing record in the Student table is updated.

4. Table Name

For a table-based DML trigger, the trigger is associated with a particular table on which the specified event occurs.

SGOU

For example:

```
ON student
```

indicates that the trigger is associated with the Student table.

If an UPDATE trigger is created on the Student table, it will fire when the specified update operation occurs on that table.

5. Trigger Level

Trigger level specifies how many times the trigger fires when the triggering event occurs. There are two commonly used trigger levels:

- **Statement-Level Trigger**
- **Row-Level Trigger**

A statement-level trigger fires once for the entire SQL statement, regardless of the number of rows affected.

A row-level trigger fires, once for each row affected by the SQL statement.

For example, if an UPDATE statement modifies the marks of 10 students:

- A statement-level trigger fires once.
- A row-level trigger fires 10 times.

The FOR EACH ROW clause is used to specify a row-level trigger.

6. Trigger Body

The trigger body contains the PL/SQL statements that are executed when the trigger is fires.

For a simple trigger, the trigger body generally contains the BEGIN and END sections and the statements that perform the required action.

For example:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student record updated.');
END;
/
```

Here, the DBMS_OUTPUT.PUT_LINE() statement represents the action performed by the trigger.

SGOU

Example of a Trigger

Let us consider a simple example of a trigger using a university Student table.

Suppose we want the database to automatically display a message whenever a student's marks are updated.

```
CREATE OR REPLACE TRIGGER marks_update
AFTER UPDATE ON student
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student marks have been updated.');
END;
/
```

In this example:

- marks_update is the name of the trigger.
- AFTER specifies the timing of the trigger.
- UPDATE is the triggering event.
- student is the table associated with the trigger.
- FOR EACH ROW specifies that the trigger is a row-level trigger.
- The statements between BEGIN and END form the trigger body.

When a student's record is updated, the trigger is automatically executed.

Executing the Trigger

The trigger fires automatically when an update operation is performed on the Student table.

For example:

```
UPDATE student
SET marks = 85
WHERE student_id = 101;
```

SGOU

When the above statement is executed, the marks_update trigger is automatically fires and displays:

Student marks have been updated.

3.4.3 Types of Triggers

Triggers can be classified in different ways based on the event causes the trigger to fire, the level at which it executes, and the time at which it executes.

- **DML Triggers** – classified based on the triggering event.
- **Statement-Level Triggers** – classified based on the trigger level.
- **Row-Level Triggers** – classified based on the trigger level.
- **BEFORE Triggers** – classified based on timing.
- **AFTER Triggers** – classified based on timing

Table 3.4.1 Classification of Triggers

Basis of Classification	Type	Meaning
Triggering Event	DML Trigger	Fires when an INSERT, UPDATE, or DELETE operation occurs
Trigger Level	Statement-Level Trigger	Fires once for the entire SQL statement.
Trigger Level	Row-Level Trigger	Fires once for each row affected by the SQL statement.
Timing	BEFORE Trigger	Fires before the triggering DML operation is completed.
Timing	AFTER Trigger	Fires after the triggering DML operation has been performed.

3.4.3.1 DML Triggers

DML triggers are triggers that fire automatically when a Data Manipulation Language (DML) operation is performed on a table.

DML statements are used to add, modify, and remove data from database tables. The three commonly used DML operations are:

- INSERT – Adds a new record to a table.
- UPDATE – Modifies existing data in a table.
- DELETE – Removes existing data from a table.

A DML trigger can be associated with one or more of these operations. When the

SGOU

specified operation occurs, the DBMS automatically fires the trigger.

1. INSERT Trigger

An INSERT trigger fires when a new record is added to a table.

For example, consider a university Student table. Whenever a new student is registered, a trigger can automatically perform an additional action, such as recording the registration activity.

For example:

```
INSERT INTO Student (student_id, name, marks)
VALUES (101, 'Anu', 85);
```

A trigger associated with the Student table can automatically fire when this statement is executed.

Example

```
CREATE OR REPLACE TRIGGER student_insert
AFTER INSERT ON student
BEGIN
    DBMS_OUTPUT.PUT_LINE('New student record added.');
END;
/
```

Here, AFTER INSERT means that the trigger fires after a new record is inserted into the Student table.

2. UPDATE Trigger

An UPDATE trigger fires when existing data in a table is modified.

For example, suppose the marks of a student are changed from 75 to 85.

```
UPDATE student
SET marks = 85
WHERE student_id = 101;
```

An UPDATE trigger can automatically perform an action after this change.

SGOU

For example:

```
CREATE OR REPLACE TRIGGER student_update
AFTER UPDATE ON student
BEGIN
   DBMS_OUTPUT.PUT_LINE('Student record updated.');
END;
/
```

Here, the trigger automatically fires after an UPDATE operation is performed on the Student table.

3. DELETE Trigger

A DELETE trigger fires when an existing record is removed from a table.

For example:

```
DELETE FROM student
WHERE student_id = 101;
```

A trigger can automatically perform an action after the student record is deleted.

Example

```
CREATE OR REPLACE TRIGGER student_delete
AFTER DELETE ON student
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student record deleted.');
END;
/
```

Here, AFTER DELETE means that the trigger fires after a record is deleted from the Student table.

3.4.3.2 Statement-Level and Row-Level Triggers

DML triggers can be classified based on the level at which they execute. Based on this classification, there are two types :

SGOU

1. Statement-Level Triggers
2. Row-Level Triggers

The main difference between these two types is how many times the trigger fires when a SQL statement affects one or more rows.

1. Statement-Level Triggers

A statement-level trigger fires once for the entire SQL statement, regardless of the number of rows are affected by the statement.

For example, consider the following SQL statement:

```
UPDATE student
SET marks = marks + 5
WHERE semester = 2;
```

Suppose this statement updates the marks of 50 students. A statement-level trigger fires only once, because only one UPDATE statement was executed.

```
CREATE OR REPLACE TRIGGER student_update
AFTER UPDATE ON student
BEGIN
   DBMS_OUTPUT.PUT_LINE('Student records have been updated.');
END;
/
```

In this example, FOR EACH ROW is not specified. Therefore, it is a statement-level trigger. If the UPDATE statement affects 50 student records, the trigger executes only once.

2. Row-Level Triggers

A row-level trigger fires once for each row affected by the SQL statement. The FOR EACH ROW clause is used to specify a row-level trigger.

Example

```
CREATE OR REPLACE TRIGGER student_update
AFTER UPDATE ON student
FOR EACH ROW
```

SGOU

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student record has been updated.');
END;
/
```

Suppose the following statement updates the marks of 50 students:

```
UPDATE student
SET marks = marks + 5
WHERE semester = 2;
```

Since 50 rows are affected, the row-level trigger executes 50 times, once for each affected student record.

3.4.3.3 BEFORE and AFTER Triggers

Triggers can also be classified based on when they execute in relation to the triggering database operation. Based on timing, there are two commonly used types:

1. **BEFORE Triggers**
2. **AFTER Triggers**

The main difference between them is whether the trigger fires before or after the triggering INSERT, UPDATE, or DELETE operation.

1. BEFORE Triggers

A BEFORE trigger fires before the triggering DML operation is completed.

For example, consider the following trigger:

```
CREATE OR REPLACE TRIGGER student_insert_check
BEFORE INSERT ON student
BEGIN
    DBMS_OUTPUT.PUT_LINE('Preparing to insert a student record.');
END;
/
```

SGOU

Here, BEFORE INSERT indicates that the trigger fires before the INSERT operation is completed on the Student table.

When the following statement is executed:

```
INSERT INTO Student
VALUES (101, 'Anu', 85);
```

the BEFORE trigger fires before the insertion is completed. Similarly, BEFORE triggers can be associated with UPDATE and DELETE operations.

Examples:

BEFORE UPDATE ON student

This specifies a trigger that fires before an update operation.

BEFORE DELETE ON student

This specifies a trigger that fires before a delete operation.

Uses of BEFORE Triggers

BEFORE triggers are useful when some action needs to be performed **before DML operation is completed.**

They can be used to:

- Validate data before it is stored.
- Check whether a particular condition is satisfied.
- Prepare or modify values before an operation, particularly in a row-level BEFORE trigger.
- Prevent an operation when a specified condition is not satisfied, for example, by raising an exception.

For example, a row-level BEFORE INSERT trigger can check whether a student's marks are within the valid range before the new record is inserted.

2. AFTER Triggers

An AFTER trigger fires after the triggering DML operation has been performed. It is commonly used when an action needs to be carried out after data has been inserted, updated, or deleted.

SGOU

Example

```
CREATE OR REPLACE TRIGGER student_update_log
AFTER UPDATE ON student
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student record has been updated.');
END;
/
```

Here, AFTER UPDATE indicates that the trigger fires after an existing student record has been updated.

When the following statement is executed:

```
UPDATE student
SET marks = 90
WHERE student_id = 101;
```

the update operation is performed first, and then the trigger fires. After the record is updated, the trigger fires.

Similarly, AFTER triggers can be associated with INSERT and DELETE operations.

Examples:

An AFTER trigger can also be associated with INSERT and DELETE operations.

AFTER INSERT ON Student

This specifies a trigger that fires after an INSERT operation on the Student table.

AFTER DELETE ON student

This trigger fires after a student record has been deleted.

Uses of AFTER Triggers

AFTER triggers are useful when an action needs to be performed after the database operation has taken place.

They can be used to:

- Maintain audit records.

SGOU

- Record changes made to data.
- Update related information.
- Perform actions after a successful database operation.

3.4.4 Creating and Managing Triggers

Triggers are database objects that are created and stored in the Oracle database. Once a trigger is created, it can automatically fire when its specified triggering event occurs.

Creating and managing triggers involves several operations. The main operations are:

1. Creating a Trigger
2. Enabling a Trigger
3. Disabling a Trigger
4. Dropping a Trigger

1. Creating a Trigger

Creating a trigger means defining a trigger in the database by specifying when it should fire, which event causes it to fire, and what action it should perform.

A trigger can be created using the CREATE TRIGGER statement. The CREATE OR REPLACE TRIGGER statement can also be used to create a new trigger or replace an existing trigger with the same name.

Syntax for Creating a Trigger

For a basic DML trigger, the general syntax is:

```
CREATE OR REPLACE TRIGGER trigger_name
timing triggering_event
ON table_name
[FOR EACH ROW]
BEGIN
    trigger_body;
END;
/
```

SGOU

Where,

- CREATE OR REPLACE TRIGGER : Creates a new trigger or replaces an existing trigger with the same name
- trigger_name : Specifies the name of the trigger
- timing : Specifies when the trigger fires, such as BEFORE or AFTER
- triggering_event : Specifies the event that activates the trigger, such as INSERT, UPDATE, or DELETE
- ON table_name: Specifies the table on which the trigger is defined
- FOR EACH ROW : Specifies that the trigger is a row-level trigger, If this clause is omitted, the DML trigger is statement-level.
- BEGIN : Marks the beginning of the executable part of the trigger body.
- trigger_body : Contains the PL/SQL statements executed when the trigger fires.
- END; : Marks the end of the trigger body
- / : A command used by tools such as SQL*Plus and SQL Developer to submit the completed CREATE TRIGGER definition for execution.

Example

Consider the following trigger:

```
CREATE OR REPLACE TRIGGER marks_update
AFTER UPDATE ON student
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student marks have been updated.');
END;
/
```

In this example:

- marks_update is the trigger name.
- AFTER specifies the timing of the trigger.
- UPDATE is the triggering event.

SGOU

- Student is the table associated with the trigger.
- FOR EACH ROW specifies that it is a row-level trigger.
- The statements between BEGIN and END form the trigger body.
- The trigger fires automatically for each row affected by the UPDATE statement.

2. Enabling and Disabling a Trigger

After a trigger is created, it can be enabled or disabled according to the requirements of the database system.

a. Enabling a Trigger

An enabled trigger is active and can fire automatically when its specified triggering event occurs.

A newly created trigger is enabled by default, unless it is explicitly created or altered in a disabled state. A previously disabled trigger can be enabled using the ALTER TRIGGER statement.

Syntax :

```
ALTER TRIGGER trigger_name ENABLE;
```

Here, trigger_name specifies the name of the trigger that needs to be enabled.

Example

```
ALTER TRIGGER student_update ENABLE;
```

This statement enables the student_update trigger.

After the trigger is enabled, it can fire automatically whenever its specified event occurs.

For example

```
UPDATE student
SET marks = 90
WHERE student_id = 101;
```

Since marks_update is an AFTER UPDATE row-level trigger, it fires automatically after the update for each affected row.

SGOU

b. Disabling a Trigger

A disabled trigger remains stored in the database but does not fire when its triggering event occurs.

Disabling a trigger is useful when the trigger is temporarily not required. The trigger is not deleted; and can be enabled again later.

Syntax:

```
ALTER TRIGGER trigger_name DISABLE;
```

Here, trigger_name specifies the name of the trigger that needs to be disabled.

Example

```
ALTER TRIGGER student_update DISABLE;
```

This statement disables the student_update trigger.

If an UPDATE operation is performed while the trigger is disabled:

```
UPDATE student
 SET marks = 95
WHERE student_id = 101;
```

The update operation is performed, but the disabled trigger does not fire.

Enabling the Trigger Again

```
ALTER TRIGGER student_update ENABLE;
```

After it is enabled, the trigger can fire automatically when its specified triggering event occurs.

3. Dropping a Trigger

A trigger may no longer be required when the database requirements change. In such cases, the trigger can be dropped from the database.

Dropping a trigger means permanently removing the trigger from the database. Once a trigger is dropped, it will no longer execute automatically when its triggering event occurs.

SGOU

Syntax for Dropping a Trigger

The general syntax is:

```
DROP TRIGGER trigger_name;
```

Here, trigger_name specifies the name of the trigger that needs to be removed.

Example

Suppose a trigger named student_update has been created on the student table.

```
DROP TRIGGER student_update;
```

This statement permanently removes the student_update trigger from the database.

After executing the statement, if an UPDATE operation is performed on the student table, the dropped trigger will not execute.

For example:

```
UPDATE student
SET marks = 95
WHERE student_id = 101;
```

The update operation can be performed, but the student_update trigger will no longer respond because it has been removed.

Dropping a Trigger vs Disabling a Trigger

It is important to understand the difference between disabling and dropping a trigger.

Table 3.4.2 Comparison between Dropping vs Disabling

Operation	Effect	Trigger remains in databse	Can it be used again
Disable	Temporarily prevents the trigger from firing	Yes	Yes, by enabling it again
Drop	Removes the trigger definition from the database	No	No, it must be created again

SGOU

Example: Disabling a Trigger

If the trigger is temporarily not required, it can be disabled:

```
ALTER TRIGGER student_update DISABLE;
```

The trigger remains in the database but does not fire.

It can later be enabled again:

```
ALTER TRIGGER student_update ENABLE;
```

Example: Dropping a Trigger

If the trigger is no longer required, it can be dropped:

```
DROP TRIGGER student_update;
```

The trigger is removed from the database. If the same trigger is required again later, it must be created again using a CREATE TRIGGER or CREATE OR REPLACE TRIGGER statement.'

3.4.5 Trigger Restrictions

Although triggers are useful for performing automatic actions in a database, they have certain restrictions and limitations.These restrictions are important because a trigger fires automatically as part of a database operation. If a trigger is not designed properly, it may cause errors, unwanted changes, or repeated execution.

The following are some important restrictions of triggers.

1. Triggers Cannot Accept Parameters Directly

Unlike stored procedures and functions, a trigger cannot be called by passing parameters directly.

For example, a procedure can be defined with a parameter:

The procedure can be called by providing a value.

EXECUTE display_student(101);

A trigger, however, is activated automatically by a specified database event. Therefore, parameters cannot be passed to a trigger in the same way as they are passed to a procedure.

SGOU

For example:

```
CREATE OR REPLACE PROCEDURE display_student(
        p_student_id NUMBER
)
AS
BEGIN
      DBMS_OUTPUT.PUT_LINE(p_student_id);
END;
/
```

```
CREATE OR REPLACE TRIGGER student_update_restriction
AFTER UPDATE ON Student
BEGIN
   DBMS_OUTPUT.PUT_LINE('Student record updated.');
END;
/
```

The trigger fires automatically when an appropriate UPDATE operation occurs. The user does not call it by passing a parameter.

2. Transaction Control Statements Cannot Normally Be Used

A trigger cannot normally contain transaction control statements such as:

COMMIT

ROLLBACK

SAVEPOINT

For example, the following type of statement should not normally be used directly inside a trigger:

```
BEGIN
    INSERT INTO student_audit VALUES (...);
    COMMIT;
END;
/
```

SGOU

The transaction is controlled by the SQL statement that caused the trigger to execute.

For example:

```
UPDATE student
SET marks = 85
WHERE student_id = 101;
```

If an AFTER UPDATE trigger fires as a result of this statement, the trigger's database operations normally form part of the same transaction.

Therefore, the trigger should not normally attempt to commit or roll back that transaction independently.

3. Row-Level Triggers Cannot Normally Query the Mutating Table

A row-level trigger fires once for each row affected by a DML operation. Suppose a row-level trigger is created on the Student table and an UPDATE operation is being performed on that same table. During the execution of the row-level trigger, the Student table is in the process of being modified.

If the trigger attempts to query the same table, Oracle can raise a mutating table error.

For example:

```
CREATE OR REPLACE TRIGGER student_check
AFTER UPDATE ON student
FOR EACH ROW
DECLARE
    v_count NUMBER;
BEGIN
    SELECT COUNT(*)
    INTO v_count
    FROM student;
END;
/
```

SGOU

Here, the trigger is a row-level trigger on the student table and it tries to query the same student table while the table is being modified. This is known as the mutating table restriction.

4. Triggers Cannot Be Called Like Procedures

A stored procedure can be executed explicitly by using a procedure call.

For example:

```
EXECUTE display_student(101);
```

A trigger does not normally work in this way. It is automatically fired by the DBMS when its specified triggering event occurs.

For example:

The trigger is not called using:

EXECUTE student_update_auto;

Instead, it fires automatically when an appropriate UPDATE operation occurs on the Student table.

For example:

UPDATE Student

SET marks = 90

WHERE student_id = 101;

```
CREATE OR REPLACE TRIGGER student_update
AFTER UPDATE ON student
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student record updated.');
END;
/
```

The trigger fires automatically when an UPDATE operation occurs on the student table.

SGOU

Summarised Overview

Triggers are special PL/SQL programs stored in the database that execute automatically when a specified event occurs. They are useful for performing automatic actions such as maintaining audit records, validating data, enforcing rules, and maintaining related information. The main components of a trigger are trigger name, timing, triggering event, table name, trigger level, and trigger body. Triggers are commonly activated by DML events such as INSERT, UPDATE, and DELETE. Based on the trigger level, they are classified as statement-level triggers, which execute once for the entire SQL statement, and row-level triggers, which execute once for each affected row using the FOR EACH ROW clause. Based on timing, BEFORE triggers execute before the DML operation, while AFTER triggers execute after the operation. A trigger is created using the CREATE OR REPLACE TRIGGER statement. Triggers can be enabled using ALTER TRIGGER ... ENABLE, disabled using ALTER TRIGGER ... DISABLE, and permanently removed using DROP TRIGGER. Unlike procedures and functions, triggers cannot accept parameters directly and are not normally called explicitly; they execute automatically in response to events. Transaction control statements such as COMMIT, ROLLBACK, and SAVEPOINT cannot normally be used directly inside a trigger. Row-level triggers also have restrictions when accessing the table currently being modified.

SGOU

Assignments

1. Define a trigger in PL/SQL. Explain the purpose of triggers and describe the main components of a trigger with suitable examples.
2. What are DML triggers? Explain INSERT, UPDATE, and DELETE triggers with suitable PL/SQL examples.
3. Explain statement-level and row-level triggers. Differentiate between them with suitable examples and explain the role of the FOR EACH ROW clause.
4. Explain BEFORE and AFTER triggers in PL/SQL. Discuss their differences and illustrate each type with suitable examples.
5. Explain the procedure for creating and managing triggers in PL/SQL. Describe the syntax of CREATE OR REPLACE TRIGGER and explain each

component with a suitable example. Also explain how triggers are enabled, disabled, and dropped.

6. What are the restrictions of triggers in PL/SQL? Explain any four important restrictions with suitable examples.

Reference

1. Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed., McGraw-Hill Education, 2020.

2. Elmasri, Ramez, and Shamkant B. Navathe. *Fundamentals of Database Systems*. 7th ed., Pearson, 2017.

3. Coronel, Carlos, and Steven Morris. *Database Systems: Design, Implementation, & Management*. 13th ed., Cengage Learning, 2019.

Suggested Reading

1. Oracle. *Oracle Database PL/SQL Language Reference*. Oracle Corporation, 2025.

2. Feuerstein, Steven. *Oracle PL/SQL Programming*. O'Reilly Media.

3. Loney, Kevin. *Oracle Database 12c: The Complete Reference*. McGraw-Hill Education.

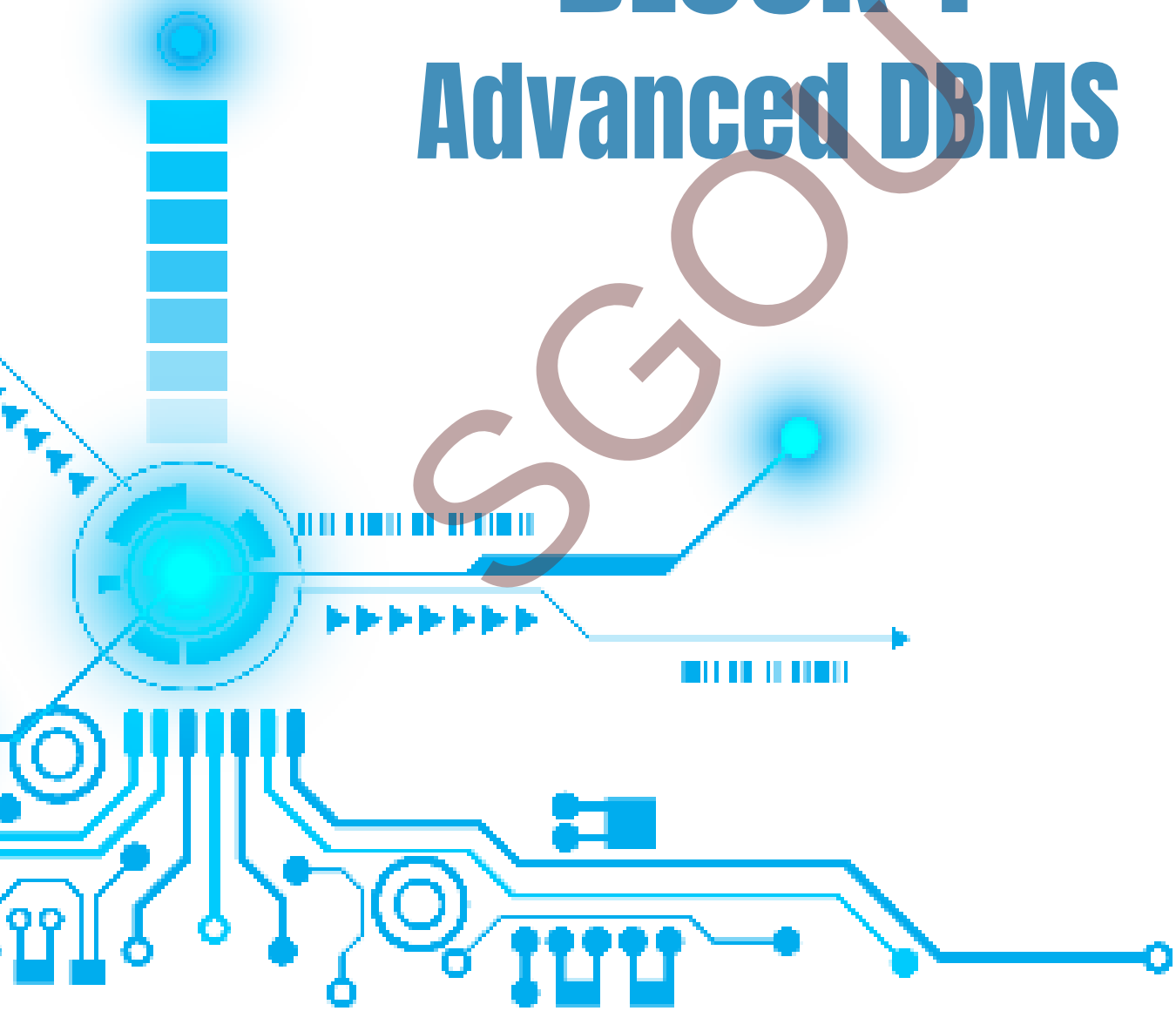
SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

BLOCK 4
Advanced DBMS



SGOU

1 UNIT

NoSQL Databases

Learning Outcomes

After completing this unit, the learner will be able to:

- define NoSQL databases and explain their key characteristics
- classify the major types of NoSQL databases
- describe Key-Value and Document databases with examples
- explain the basic architecture and data model of Apache Cassandra
- describe Graph databases and explain the basic concepts of Neo4j

Background

Every day, we interact with applications that generate enormous amounts of data. Social media platforms record posts, comments, and connections; online shopping applications store product details and customer activities; mobile applications generate user profiles and activity logs; and educational platforms manage learner information, courses, assessments, and interactions.

But an important question arises:

Can all these different types of data be efficiently managed using traditional relational databases?

Traditional relational databases work very well when data is structured, relationships are clearly defined, and the database schema is relatively stable. However, modern applications often handle large volumes of diverse data, including semi-structured

SGOU

and unstructured data. The structure of this data may also change frequently as applications evolve.

This creates a need for database systems that can provide flexible data models, scalability, and efficient handling of different types of data.

This is where NoSQL databases become important. NoSQL databases provide alternatives to the traditional relational model and support different ways of organizing data, such as key-value pairs, documents, wide-column data, and graphs. These models allow applications to store and retrieve data according to their specific requirements.

In this unit, you will explore what NoSQL databases are, why they are used, their major characteristics, and their different data models. You will also learn about Key-Value and Document databases, Apache Cassandra, and Graph databases using Neo4j. By the end of the unit, you will understand how different NoSQL database models are used to address the requirements of modern applications.

Keywords

NoSQL Databases, Key-Value Stores, Document Stores, Column-Family Stores, Graph Databases

Discussion

SGOU

4.1.1 Introduction to NoSQL Databases

Modern applications generate and process a huge amount of data every day. Social media platforms, online shopping websites, mobile applications, banking systems, and educational platforms continuously create different types of data. Some data may be highly structured, while other data may include text, images, videos, user profiles, logs, and other information that may not fit easily into a traditional table-based structure.

Traditional relational databases store data mainly in tables consisting of rows and columns. They are very useful when data has a well-defined structure and relationships. However, as applications grow and the amount and variety of data increase, managing such data using only traditional relational database structures can sometimes become challenging.

For example, consider a university application that stores information about learners. A relational database can store basic details such as learner ID, name, programme, semester, and marks in tables. However, the application may also need to store additional information such as learner preferences, activity logs, notifications, uploaded documents, or other data whose structure may change frequently.

To handle such requirements, NoSQL databases provide alternative approaches to storing and managing data.

Modern applications generate and process a huge amount of data every day. Social media platforms, online shopping websites, mobile applications, banking systems, and educational platforms continuously create different types of data. Some data may be highly structured, while other data may include text, images, videos, user profiles, logs, and other information that may not fit easily into a traditional table-based structure.

Traditional relational databases store data mainly in tables consisting of rows and columns. They are very useful when data has a well-defined structure and relationships. However, as applications grow and the amount and variety of data increase, managing such data using only traditional relational database structures can sometimes become challenging.

For example, consider a university application that stores information about learners. A relational database can store basic details such as learner ID, name, programme, semester, and marks in tables. However, the application may also need to store additional information such as learner preferences, activity logs, notifications, uploaded documents, or other data whose structure may change frequently.

To handle such requirements, NoSQL databases provide alternative approaches to storing and managing data.

4.1.2 Basics of NoSQL Databases

NoSQL databases are database management systems designed to store and process data without depending mainly on the traditional table-based structure used by relational databases. The term NoSQL is commonly understood as “Not Only SQL”, indicating that these databases can support different ways of storing and accessing data in addition to traditional relational approaches.

In a relational database, data is generally organized into tables with predefined columns and relationships. In a NoSQL database, the structure of data depends on the type of NoSQL database being used. This allows applications to work with data that may be large, varied, or frequently changing.

For example, consider a university application. Information about a learner may include learner ID, name, programme, contact details, courses, interests, login activities, and other information. Some learners may have additional information that is not available for others. A NoSQL database can store such information more flexibly without requiring the same fixed structure for every record. NoSQL databases are database management systems designed to store and process data using models other than the

SGOU

traditional relational table-based model. The term NoSQL is commonly understood as "Not Only SQL," indicating that these databases provide alternatives to the traditional relational approach for storing and accessing data.

In a relational database, data is generally organized into tables with predefined columns and relationships. In a NoSQL database, the structure and representation of data depend on the type of NoSQL database being used. This provides flexibility for applications that work with data that may be large, diverse, or frequently changing.

For example, consider a university application. Information about a learner may include learner ID, name, programme, contact details, courses, interests, login activities, and other information. Some learners may have additional information that is not available for others. A NoSQL database can represent such information more flexibly without requiring every record to have exactly the same structure.

Main Characteristics of NoSQL Databases

The important characteristics of NoSQL databases include:

- Flexible Data Structure: NoSQL databases can store data without requiring a fixed table structure in many cases.
- High Scalability: They can handle increasing amounts of data and users by scaling the database system, including through distributed or horizontal scaling.
- Large-Scale Data Handling: They are suitable for applications that generate and process large volumes of data.
- High Performance: NoSQL databases can provide efficient data access for specific application requirements and workloads.
- Distributed Data Storage: Data can be distributed across multiple servers or locations.
- Support for Different Data Models: NoSQL databases support different models, including key-value, document, wide-column, and graph models.
- Suitable for Changing Data: They are useful for applications in which the structure or attributes of data may change frequently.How Does a NoSQL Database Store Data?

Unlike a relational database, there is no single method of storing data in NoSQL databases. The storage and representation method depends on the type of NoSQL database being used. For example:

Table 4.1.1 NoSQL Type

NoSQL Type	Data Representation	Example
Key-Value Store	Key and value pairs	Amazon DynamoDB

SGOU

Document Store	Documents such as JSON-like documents	MongoDB
Column-Family Store	Data organised into column families	Apache Cassandra
Graph Database	Nodes and relationships	Neo4j

4.1.3 Benefits of NoSQL Databases

NoSQL databases provide several benefits for modern applications that handle large amounts of data, changing data structures, and a large number of users. Their flexible design makes them suitable for applications where traditional relational database structures may not always be convenient.

The major benefits of NoSQL databases are explained below.

1.Flexible Data Structure

NoSQL databases generally allow data to be stored without requiring every record to have exactly the same structure. New fields can often be added without modifying a fixed table schema.

For example, one learner document may contain an email address, while another learner document may additionally contain a phone number and address.

This flexibility is useful when the type or structure of information stored by an application changes frequently.

2.High Scalability

Many NoSQL databases are designed to handle increasing amounts of data and growing numbers of users. They commonly support **horizontal scaling**, in which additional servers or nodes can be added to distribute the workload.

For example, an online university application with a growing number of learners may add additional database servers or nodes to handle increased workloads.

3.Suitable for Large Volumes of Data

Modern applications can generate large amounts of data from websites, mobile applications, social media platforms, sensors, and transaction systems. Many NoSQL databases are designed to handle such large-scale workloads across distributed systems.

For example, a university may generate large amounts of data from learner activities, examination records, login information, and application logs.

4.Efficient Data Access

NoSQL databases can provide efficient data access for applications that require frequent reading and writing of data. Their data models can be designed according to the specific requirements of an application.

SGOU

For example, a mobile application that frequently retrieves learner profile information can use a document or key-value database to provide efficient access to the required data.

5. Distributed Data Storage

Many NoSQL databases support **distributed data storage**, in which data is stored across multiple servers or nodes. This can help distribute workloads and support scalability and availability.

Some NoSQL systems also use **replication**, where copies of data are maintained on multiple nodes. This can allow the system to continue serving data when a node becomes unavailable, depending on the database configuration.

6. High Performance

NoSQL databases can provide fast data access for applications that require frequent reading and writing of data. Their data models can be designed according to the specific requirements of an application.

For example, a mobile application that frequently retrieves learner profile information can use a NoSQL database to provide quick access to the required data.

7.Support for Different Data Types

NoSQL databases support different data models, allowing developers to select a suitable model according to the requirements of an application.

Common NoSQL models include:

- **Key-Value**
- **Document**
- **Column-Family**
- **Graph**

For example, a key-value database may be suitable for simple key-based lookups, while a graph database may be suitable for applications involving complex relationships between data items.

4.1.4 Challenges of NoSQL Databases

The major challenges of NoSQL databases are explained below.

1. Lack of a Standardized Query Language

Relational databases commonly use SQL as a widely standardized language for accessing and managing data. NoSQL databases, however, may use different query languages, APIs, or query mechanisms depending on the database system.

SGOU

For example, MongoDB, Apache Cassandra, and Amazon DynamoDB use different approaches for querying and manipulating data. Cassandra uses CQL (Cassandra Query Language), while MongoDB uses its own query and aggregation mechanisms.

Therefore, learning one NoSQL database does not necessarily mean that the same query methods can be directly used with another NoSQL database.

2. Flexible Schema Can Increase Complexity

The flexibility of NoSQL databases is useful, but it can also create challenges. Since records may have different structures, maintaining consistency in the organization and representation of data can become more difficult.

For example, if different learner records use different field names or formats for storing contact information, managing and processing the data may become difficult.

Therefore, applications often need appropriate data validation and design practices to maintain consistency.

3. Limited Support for Complex Relationships in Some NoSQL Models

Some NoSQL databases are not primarily designed to handle complex relationships between data items in the same way as relational or graph-oriented systems.

For example, if a university application needs to represent complex relationships among learners, courses, teachers, and departments, selecting an appropriate NoSQL model becomes important.

A Graph database, such as Neo4j, is specifically designed to represent and query relationships. Therefore, the suitability of NoSQL for relationship-heavy applications depends on the particular NoSQL model being used.

4. Data Consistency Considerations

Some distributed NoSQL systems are designed to provide high availability and scalability while allowing certain consistency trade-offs. In such systems, applications may need to consider situations in which updated data is temporarily not identical across different replicas.

For example, when learner information is updated, the updated information may not become immediately visible at every replica in a system that uses eventual consistency.

However, this is not a limitation of all NoSQL databases. Consistency guarantees vary among different NoSQL systems and configurations.

5. Different Data-Modelling Approaches

NoSQL databases may require a different approach to data modelling compared with relational databases.

In relational databases, data is commonly normalized into multiple related tables. In NoSQL databases, data may instead be embedded, denormalized, partitioned, or organized according to application access patterns.

SGOU

For example, a document database may store a learner's basic information and related course information together in a single document.

Therefore, designing an efficient NoSQL database requires understanding how the application will store, retrieve, and process the data.

4.1.5 Types of NoSQL Databases

NoSQL databases use different methods to organize and store data. Unlike relational databases, which mainly organize data in tables with predefined structures, NoSQL databases can represent data in different forms depending on the requirements of an application.

The four major types of NoSQL databases covered in this unit are:

1. Key-Value Stores
2. Document Stores
3. Wide-Column Stores (Column-Family Stores)
4. Graph Databases

4.1.5.1 Key-Value Stores

A Key-Value Store is a type of NoSQL database in which data is represented as a collection of key-value pairs. The key is used to identify and locate an item, while the value contains the information associated with that key.

A simple example is:

Table 4.1.2 Example of a Key-Value Store

Key	Value
101	Anu
102	Rahul
103	Meera

Here, 101, 102, and 103 are keys, while Anu, Rahul, and Meera are the corresponding values.

Characteristics of Key-Value Stores

1. Simple Data Model

Data is represented as key-value pairs, where each key identifies an item. This makes the basic data model simple and easy to understand.

SGOU

2. Fast Data Access

When the required key is known, the database can efficiently locate the corresponding item.

For example:

Key: 102

↓

Value: Rahul

3.Flexible Values

The value associated with a key may contain simple data or a more complex collection of attributes, depending on the database system.

For example:

Key: 101

Value:

{

Name: "Anu",

Course: "B.Tech",

Marks: 85

}

4. Easy to Scale

Many key-value databases support distributed storage and horizontal scaling, allowing additional servers or nodes to be used as the amount of data and workload increases.

5. Suitable for Large-Scale Applications

Key-value databases can be designed to handle large amounts of data and high volumes of requests, making them suitable for many large-scale applications.

6. Useful for Key-Based Access

They are particularly useful when data is frequently accessed using a known unique identifier.

For example:

User ID → User Profile

Student ID → Student Details

Product ID → Product Details

SGOU

Amazon DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service provided by Amazon Web Services (AWS). It supports both key-value and document data models.

DynamoDB is designed to provide scalable and efficient data access for applications that require large-scale data storage and high request volumes.

In the key-value model, each item is identified using its primary key. The primary key is used to locate the corresponding item efficiently.

Example

Consider a Student table in DynamoDB.

Table 4.1.3 Student table

StudentID	Name	Programme	Semester
101	Anu	MCA	2
102	Rahul	BCA	4
103	Meera	Bsc Data Science	2

Here:

- Student_ID can be used as the primary key.
- Each row represents an item.
- Name, Course, and Marks are attributes of the item.
- The Student_ID is used to locate a particular student item.

For example, if the application searches for StudentID = 102, the corresponding information about Rahul can be retrieved using the key.

4.1.5.2 Document Stores

A Document Store is a type of NoSQL database that stores data in the form of documents rather than organizing it primarily into traditional relational rows and columns. A document contains related information about an entity and is commonly represented using JSON-like formats.

For example, MongoDB stores documents using BSON (Binary JSON), which supports JSON-like document structures along with additional data types.

Each document can have a flexible structure, so different documents in the same collection may contain different fields.

Example

Consider the following learner document:

SGOU

```
{
   "student_id": 101,
   "name": "Anu",
   "programme": "MCA",
   "semester": 2
}
```

Another learner may have additional information:

```
{
   "student_id": 102,
   "name": "Rahul",
   "programme": "BCA",
   "semester": 4,
   "skills": ["Python", "Java"]
}
```

Here, the second document contains a skills field that is not present in the first document. This demonstrates the flexible schema of document databases.

The documents can also contain nested documents and arrays, allowing related information to be represented together.

Structure of a Document Store

A document store generally organizes data using databases, collections, documents, and fields as shown in figure 4.1.1.

The different parts of this structure can be understood as follows:

- Database: A database contains one or more collections.
- Collection: A collection is a group of related documents. It is broadly comparable to a table in a relational database, although it does not require every document to have the same structure.
- Document: A document stores information about a particular entity and may contain fields, nested documents, and arrays.
- Field: A field represents an individual piece of information within a document, such as student ID, name, or programme.

SGOU

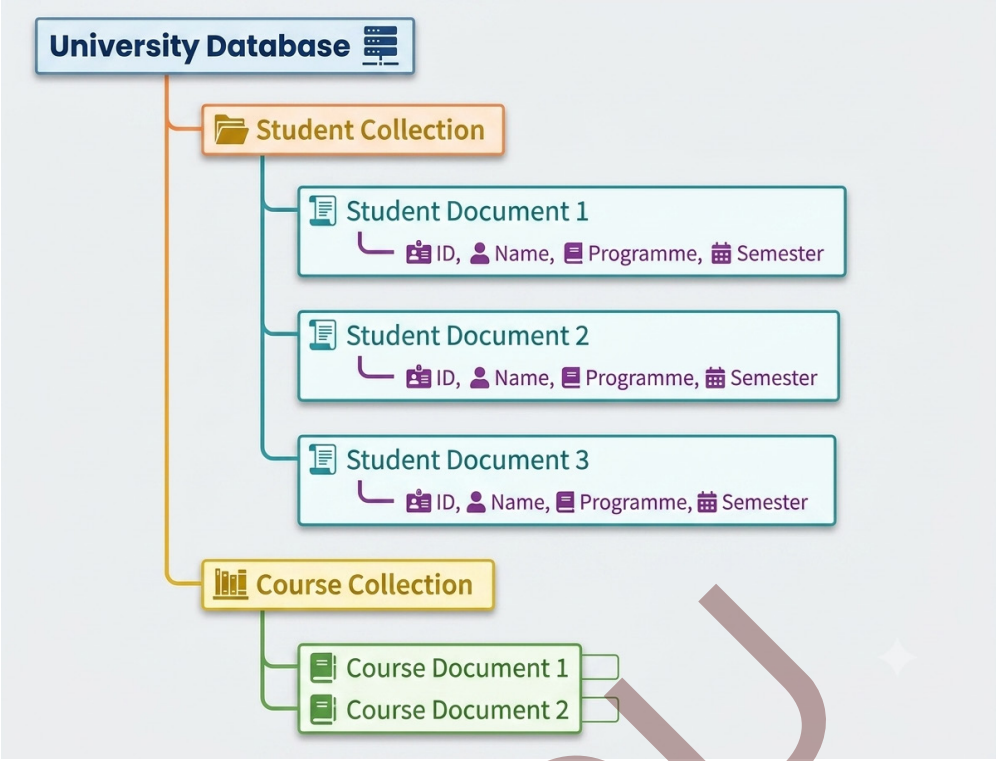


Fig. 4.1.1 Structure of Document Store

MongoDB

MongoDB is a popular document-oriented NoSQL database. It stores data as documents within collections and provides a flexible structure for storing different types of information.

For example, an online shopping application can use a Product collection to store information about different products. Each product can be represented as a separate document. Different product documents can contain different fields depending on the information available for each product.

MongoDB stores documents internally in BSON (Binary JSON) format. BSON is a binary representation of JSON-like documents and supports additional data types such as dates and binary data.

The basic organization of MongoDB can be represented as in the figure 4.1.2.

Characteristics of Document Stores

1. Document-Based Storage:

Data is stored as individual documents rather than being organized primarily into traditional rows and columns. Each document contains related information about an entity.

SGOU

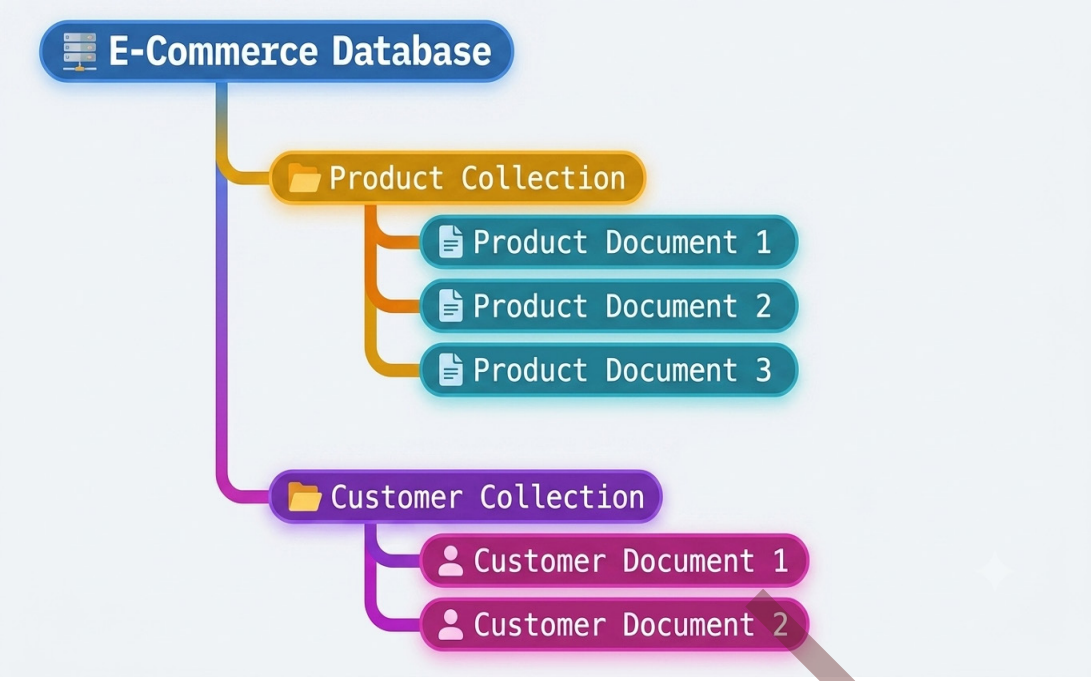


Fig. 4.1.2 Basic Structure of MongoDB

2.Flexible Data Structure:

Documents within the same collection can have different fields. This allows applications to store information even when the structure of the data changes.

3. Support for Nested Data:

Documents can contain nested objects and arrays. This allows related information to be stored together within a single document.

4. Support for Semi-Structured Data:

Document stores can handle data that does not follow a rigid fixed structure, making them suitable for many modern applications.

5. Easy Data Representation:

Documents are commonly represented using JSON-like structures, which are relatively easy for developers to understand and work with.

6. Scalability:

Many document databases support distributed storage and horizontal scaling, allowing them to handle increasing amounts of data and workloads.

7. Suitable for Changing Applications:

New fields can often be added to documents as application requirements change without requiring all existing documents to have exactly the same structure.

SGOU

Applications of Document Stores

Document stores are commonly used in:

- **E-Commerce Applications:** Storing product and customer information.
- **User Profile Management:** Storing user details, preferences, and settings.
- **Content Management Systems:** Managing articles, posts, and other content.
- **Mobile Applications:** Storing application data with varying structures.
- **Web Applications:** Managing user-generated and semi-structured data.
- **Real-Time Applications:** Handling frequently changing application data.

4.1.5.3 Column-Family Stores

A Column-Family Store, also known as a Wide-Column Store, is a type of NoSQL database designed to organize and process large volumes of data across multiple servers.

Unlike traditional relational databases, which commonly organize data into fixed tables and use relationships between tables, column-family stores are designed for distributed storage, horizontal scalability, and high-volume data processing.

In a column-family database, data is organized using rows and columns, and related columns are grouped within a column family or table. Each row is identified using a unique row key. In systems such as Apache Cassandra, data is distributed across multiple nodes based primarily on the partition key.

Example

Consider the following course-enrolment information:

CourseID	LearnerID	CourseName	Programme	Semester
C101	1001	Database Management Systems	MCA	2
C101	1002	Database Management Systems	MCA	2
C102	1003	Machine Learning	MCA	3
C103	1004	Data Visualization	BSc Data Science	2

Fig. 4.1.3 Example of a Column-Family Store

SGOU

Here:

- CourseID identifies the course.
- LearnerID identifies the learner.
- CourseName, Programme, and Semester are columns containing information about the enrolment.
- A suitable row key or partition key can be selected based on how the application needs to retrieve the data.
- The data can be partitioned and distributed across multiple servers in a cluster.

The main purpose of a column-family store is not simply to store data in columns. It is designed to efficiently manage large-scale distributed data and support applications with high volumes of read and write operations.

Characteristics of Column-Family Stores

1. Column-Family Data Model

Data is organized into rows and columns, with columns defined within a table or column family. The model is designed to efficiently organize large datasets for distributed processing.

2.Flexible Data Representation

Column-family databases can provide flexibility in how data is represented and accessed. Some systems also support collections or other structures for representing varying information.

3. High Scalability

Column-family databases are designed to support horizontal scaling. Additional servers or nodes can be added when the amount of data or workload increases.

4. High Availability

Many column-family databases support data replication across multiple nodes. This can improve availability and allow the system to continue operating when a node becomes unavailable, depending on the configuration.

5. Efficient Read and Write Operations

Column-family stores can provide efficient read and write operations for workloads for which the data model and access patterns are appropriately designed. They are commonly used in applications that handle high volumes of data.

6. Distributed Data Storage

Large datasets can be partitioned and distributed across multiple nodes in a cluster. This allows the database to manage data that may be too large or demanding for a single server.

SGOU

7. Query-Oriented Data Design

Tables are generally designed according to the queries and access patterns required by the application. This is an important principle in systems such as Cassandra.

For example, if an application frequently retrieves all courses taken by a particular learner, the table can be designed specifically to support that query efficiently.

8. Suitable for Large Datasets

Column-family stores are particularly suitable for applications that generate and process large volumes of distributed data, such as sensor data, activity logs, time-series workloads, and large-scale transactional workloads.

Apache Cassandra

Apache Cassandra is a popular open-source wide-column NoSQL database designed for distributed data storage, high availability, and large-scale data processing.

Cassandra uses a distributed and decentralized architecture in which data can be stored across multiple servers, called nodes. Instead of depending on a single central server, Cassandra distributes data across the cluster and can replicate it across multiple nodes.

This architecture allows the system to continue operating even when some nodes become unavailable, depending on the replication and consistency configuration.

Cassandra is particularly suitable for applications that require:

- Large-Scale Data Storage
- High Availability
- High Write Throughput
- Fast Data Access
- Horizontal Scalability
- Distributed Data Processing

Cassandra organizes data using concepts such as clusters, nodes, keyspaces, tables, partitions, rows, and columns.

Structure of Apache Cassandra

The basic organization of data in Cassandra can be represented as:

Cluster → Keyspace → Table → Partition → Row → Columns

1. Cluster

A cluster is a collection of connected Cassandra nodes that work together to store and manage data.

SGOU

Cassandra distributes data across the nodes in the cluster. Data can also be replicated across multiple nodes according to the configured replication factor. This helps improve availability and fault tolerance.

For example:

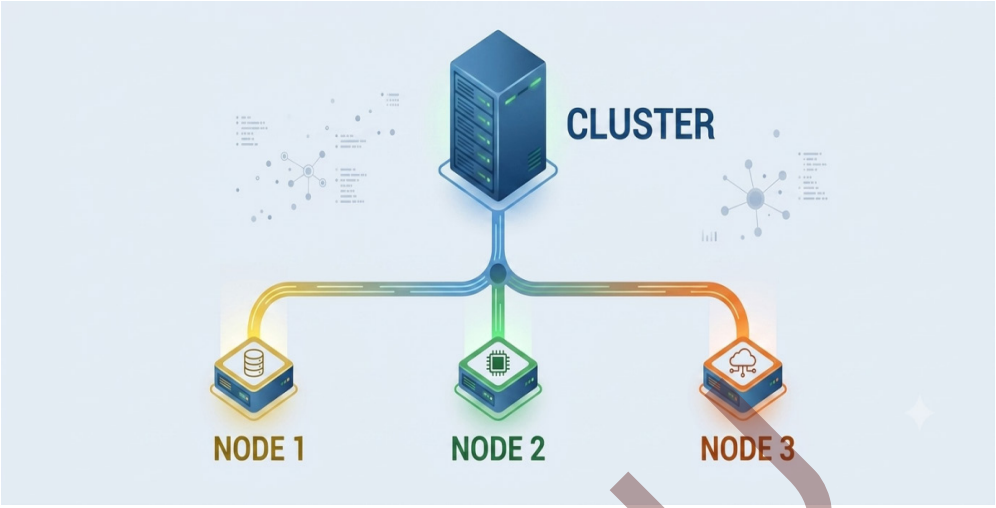


Fig. 4.1.4 Cluster in Apache Cassandra

Here, Node 1, Node 2, and Node 3 are individual servers participating in the Cassandra cluster.

If data is replicated across multiple nodes, more than one node may contain copies of the same data. Therefore, if one node fails, the data may still be available from another node that holds a replica.

2. Node

A node is an individual server that forms part of a Cassandra cluster. Each node is responsible for storing a portion of the data and may also store replicas of data belonging to other nodes.

For example:

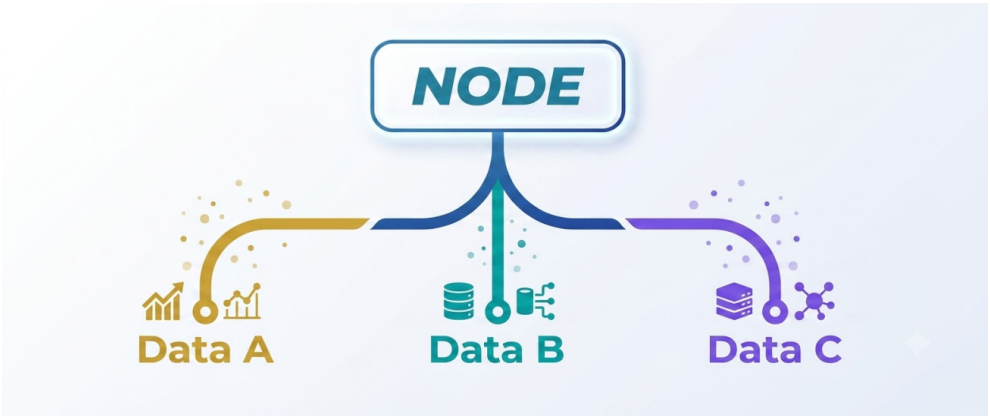


Fig. 4.1.5 Node in Apache Cassandra

SGOU

The distribution of data among nodes allows Cassandra to handle large datasets without requiring all data to be stored on a single server.

3. Keyspace

A keyspace is a logical namespace in Cassandra that contains one or more tables. It also defines important settings such as the replication strategy and replication factor.

For example:

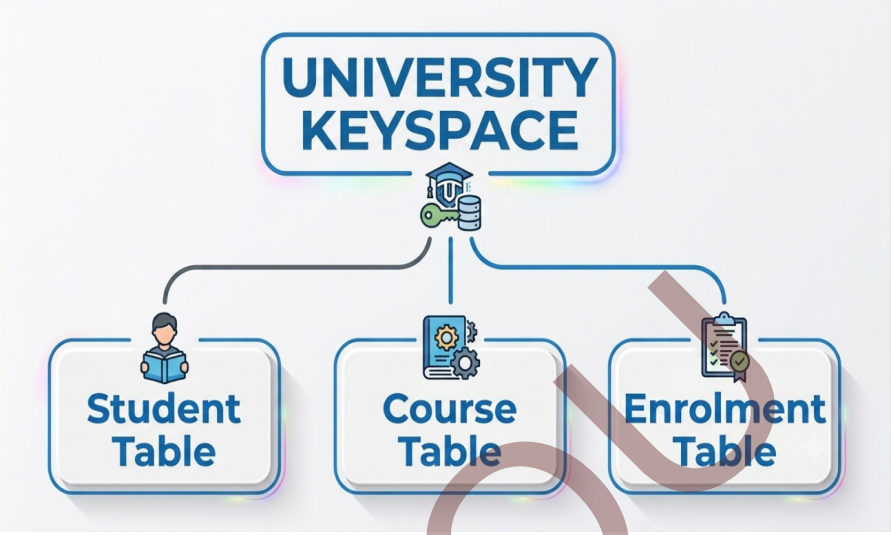


Fig. 4.1.6 Keyspace in Apache Cassandra

4. Table

A table stores data in rows and columns. Cassandra tables are designed according to the queries and access patterns of the application.

For example:

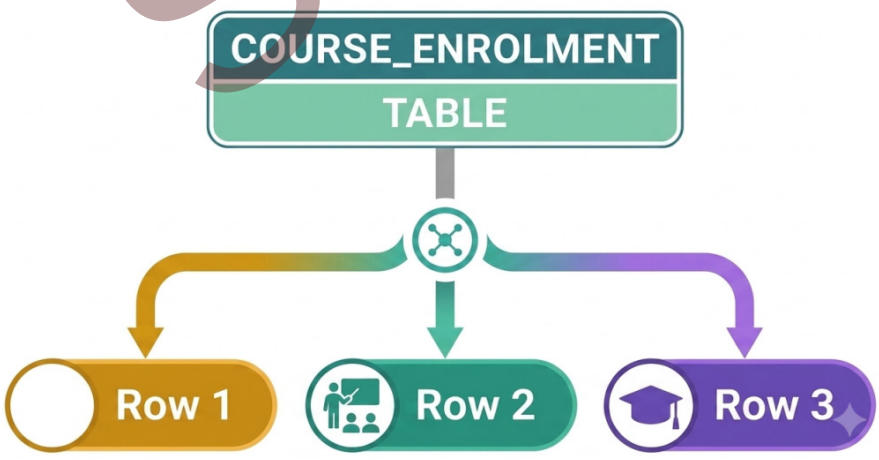


Fig. 4.1.7 Table in Apache Cassandra

SGOU

5. Partition

A partition is a group of related rows that are stored together based on the partition key.

The partition key determines how data is distributed across the Cassandra cluster.

For example, if Student_ID is selected as the partition key, different student records can be distributed across different nodes.

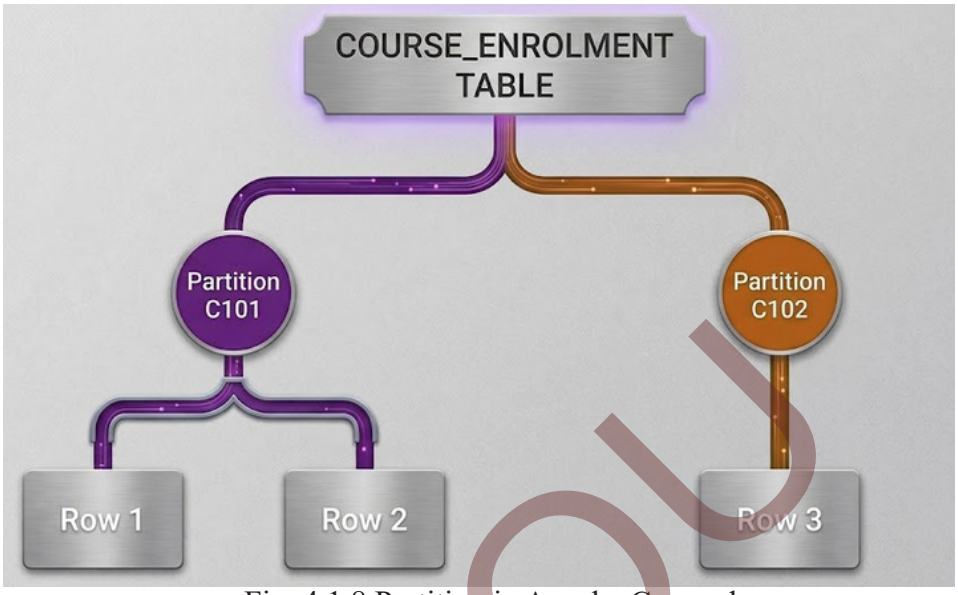


Fig. 4.1.8 Partition in Apache Cassandra

Here, the two rows having CourseID = C101 belong to the same partition.

6. Row

A **row** represents an individual record within a partition. It contains a set of columns that describe the stored data.

7. Columns

Columns represent individual attributes of a row.

For example:

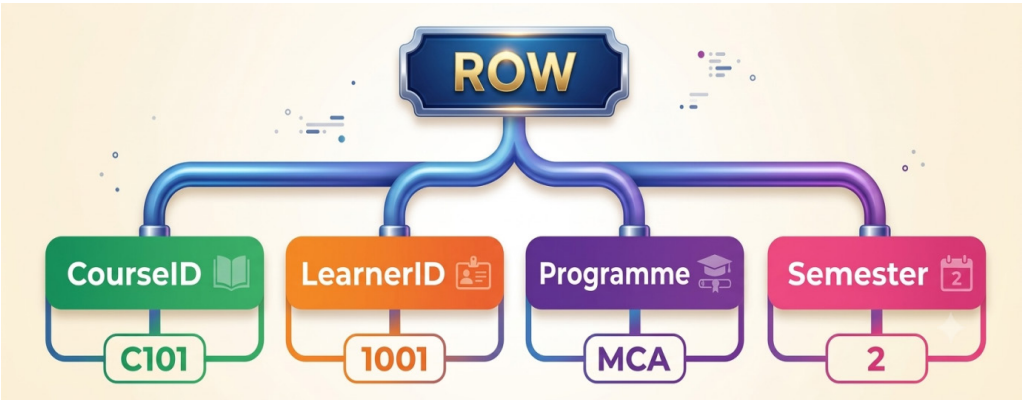


Fig. 4.1.9 Columns in Apache Cassandra

SGOU

Here, Student_ID, Name, Programme, and Semester are columns.

Cassandra Data Hierarchy

Table 4.1.4 Components of Apache Cassandra

Component	Description
Cluster	Collection of Cassandra nodes working together
Node	Individual server participating in the cluster
Keyspace	Logical namespace containing tables
Table	Stores data as rows and columns
Partition	Group of rows determined by the partition key
Row	Individual record within a partition
Column	Individual attribute of a row

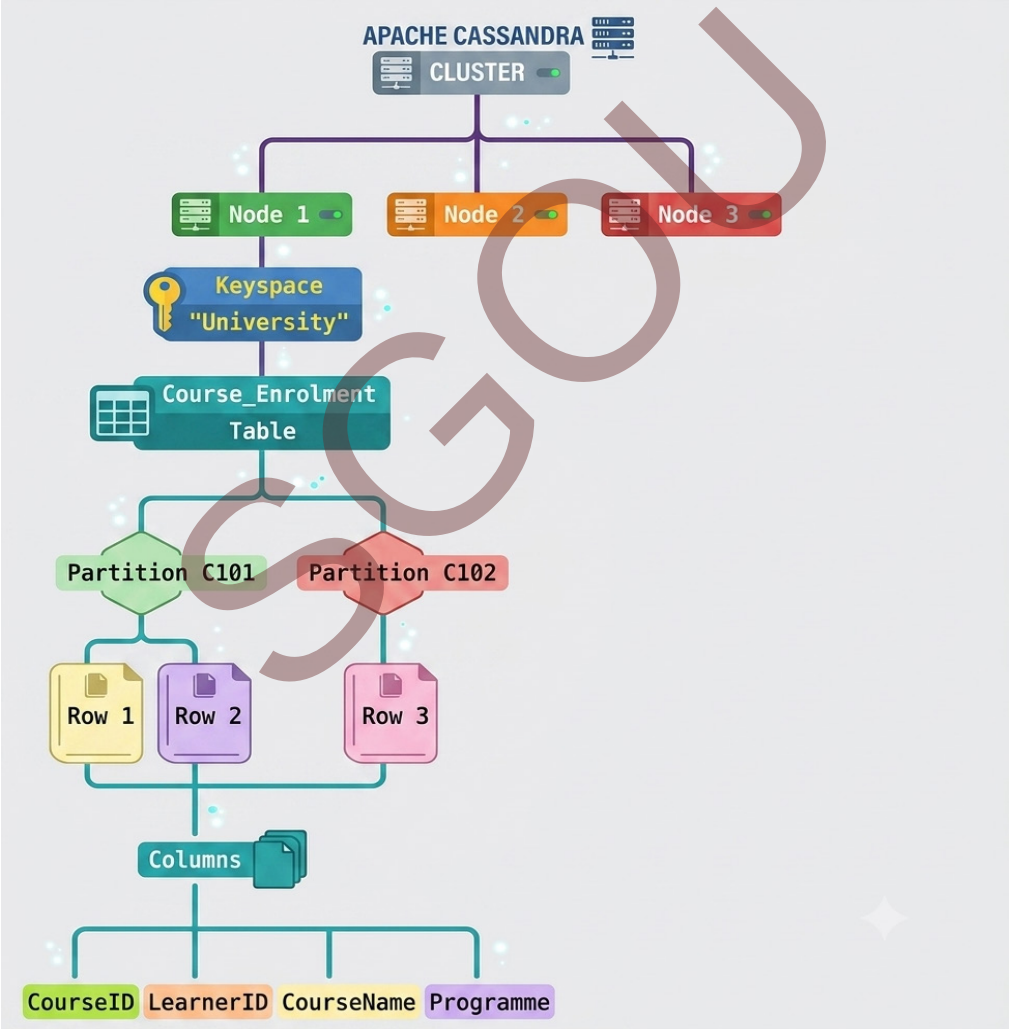


Fig. 4.1.10 Complete Structure of Apache Cassandra

4.1.5.4 Graph Databases

A Graph Database is a type of NoSQL database that represents and stores data using nodes, relationships, and properties. Unlike relational databases, which primarily organize data into tables consisting of rows and columns, graph databases focus on the connections and relationships between data entities.

In a graph database:

- Nodes represent entities or objects.
- Relationships represent connections between entities.
- Properties provide additional information about nodes and relationships.

This structure makes graph databases particularly suitable for applications in which relationships between data are important.

For example, in a university application, learners, courses, and instructors can be represented as nodes, while relationships such as ENROLLED_IN and TAUGHT_BY can represent their connections.

Example

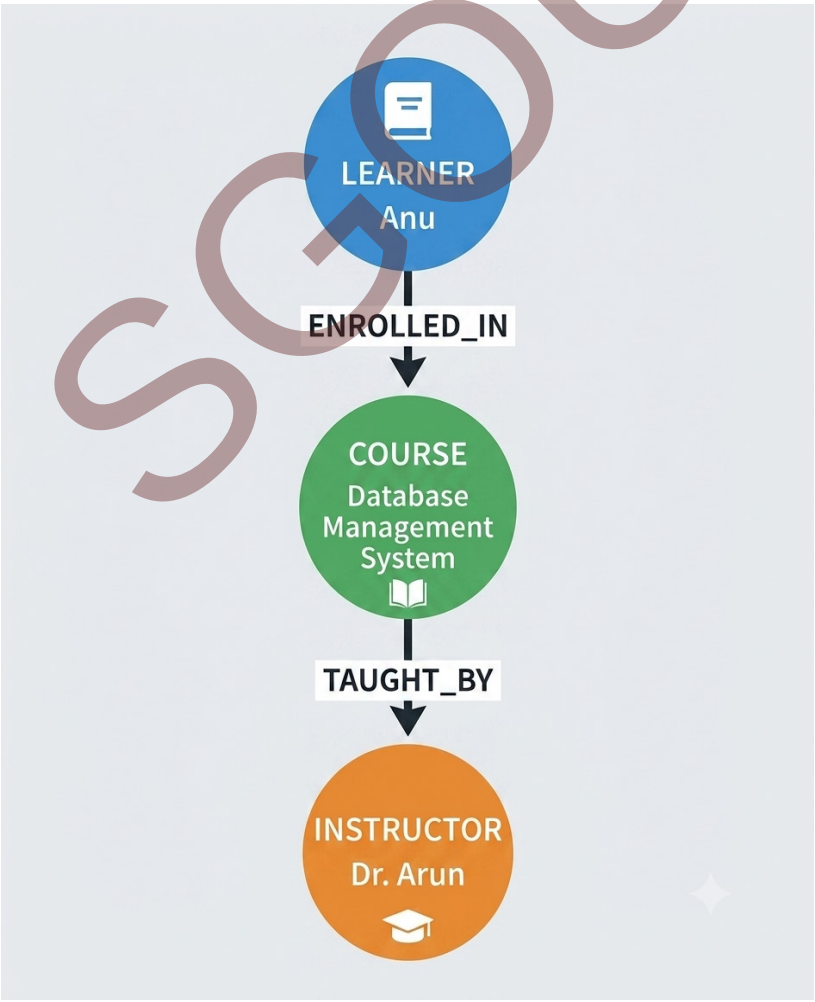


Fig. 4.1.11 Example of Nodes and Relationships in a Graph Database

SGOU

Structure of a Graph Database

A graph database represents data using **nodes, relationships, and properties**. These components work together to represent entities and the connections between them.

The structure of a graph database can be understood through the following components.

1. Node

A node represents an individual entity or object in a graph database. A node can represent a person, product, course, organization, location, or any other entity.

For example, in a university application as in Fig. 4.1.11

- Anu can represent a learner node.
- Database Management Systems can represent a course node.
- Dr. Arun can represent an instructor node.

Each node represents a distinct entity, while the connections between the nodes represent relationships among these entities.

2. Relationship

A relationship represents a connection between two nodes in a graph database. It describes how one entity is associated with another entity. Relationships are an important part of graph databases because they explicitly represent the connections among different entities.

As shown in Fig. 4.1.11

Anu ── ENROLLED_IN ──→ Database Management Systems

Database Management Systems ── TAUGHT_BY ──→ Dr. Arun

Here:

- ENROLLED_IN connects the learner to the course.
- TAUGHT_BY connects the course to the instructor.

Relationships can also contain properties when additional information about the connection needs to be stored.

3. Property

A property is an attribute that provides additional information about a node or relationship. Properties are generally represented as key-value pairs.

As shown in Fig. 4.1.11, the nodes representing the learner, course, and instructor can contain properties that provide additional details about them.

For example, the Anu node may contain the following properties:

SGOU

Property	Value
StudentID	101
Name	Anu
Programme	MCA
Semester	2

A relationship can also have properties. For example, the ENROLLED_IN relationship could contain a property such as:

Property	Value
AcademicYear	2026
Grade	A

Thus, properties can provide additional information about both entities and their connections.

Characteristics of Graph Databases

The important characteristics of graph databases are as follows:

1. Relationship-Oriented

Graph databases are designed to represent and manage relationships between entities efficiently. Relationships are stored explicitly as part of the graph.

2. Flexible Data Structure

Graph databases provide a flexible structure in which different nodes and relationships can have different properties according to application requirements.

3. Efficient Relationship Traversal

Related data can be accessed by traversing relationships between nodes. This makes graph databases suitable for applications that require analysis of connected data. A graph database can follow these connections to retrieve related information.

4. Support for Highly Connected Data

Graph databases are particularly suitable for applications in which a large number of entities are strongly interconnected.

5. Natural Representation of Data

Real-world entities and their relationships can be represented naturally using nodes and relationships.

SGOU

For example:

Person → Works_For → Company

Person → Lives_In → City

Person → Knows → Person

6. Efficient Access to Connected Data

Graph databases can efficiently retrieve related information by traversing the connections between nodes, particularly for queries involving multiple relationships.

7. Schema Flexibility

Graph databases can accommodate evolving data structures by allowing new properties, node types, and relationships to be introduced as application requirements change.

8. Suitable for Complex Relationships

Graph databases are well suited for applications involving multiple and complex relationships among entities.

Examples include:

- Social networks
- Recommendation systems
- Fraud detection
- Knowledge graphs
- Network and IT infrastructure
- Transportation and route networks

Neo4j

Neo4j is a popular graph database management system designed to store, manage, and analyze highly connected data. It uses a property graph model, in which data entities are represented as nodes, connections between entities are represented as relationships, and additional information is stored as properties.

Unlike traditional relational databases, Neo4j focuses on the relationships between data as well as the data itself. This makes it suitable for applications that require efficient exploration of connections among different entities.

Neo4j provides a flexible structure for storing data because new nodes, relationships, and properties can be added as application requirements evolve. This makes it suitable for applications involving complex or changing relationships.

Neo4j uses Cypher, a declarative query language, to create, retrieve, update, and delete graph data. Cypher allows users to describe patterns in a graph and retrieve information by specifying the relationships between nodes.

SGOU

Applications of Neo4j

Neo4j is commonly used in:

1. Social Networking – Analyzing connections among users, groups, and communities.

2. Recommendation Systems – Identifying relationships among users, products, movies, or services to generate relevant recommendations.

3. Fraud Detection – Analyzing complex connections among accounts, transactions, users, and other entities to identify potentially suspicious patterns.

4. Knowledge Graphs – Representing relationships among people, organizations, concepts, and other entities.

5. Network Management – Analyzing connections among network devices, systems, and services.

Thus, Neo4j is particularly useful when relationships between data are as important as the data itself. Its property graph model and Cypher query language make it suitable for storing and querying highly connected information.

SGOU

Summarised Overview

NoSQL databases, commonly understood as "Not Only SQL," are database management systems designed to store and process large volumes of data using models other than the traditional relational table model. They are particularly useful for applications that require flexible data structures, scalability, distributed storage, and support for diverse or changing data.

The four major types of NoSQL databases are Key-Value Stores, Document Stores, Column-Family (Wide-Column) Stores, and Graph Databases. Key-Value Stores represent data as key-value pairs and are particularly suitable for efficient retrieval when the key is known. Amazon DynamoDB is an example that supports both key-value and document data models.

Document Stores represent data as documents, commonly using JSON-like structures. They organize data into databases, collections, documents, and fields. MongoDB is a popular document-oriented database that stores documents internally in BSON (Binary JSON) format.

Column-Family (Wide-Column) Stores organize data using rows and columns and are designed for distributed, large-scale data processing. They are suitable for workloads requiring scalability, high availability, and high-volume read and write operations. Apache Cassandra is a widely used example, with its basic data organization represented as Cluster → Keyspace → Table → Partition → Row → Columns.

Graph Databases represent data using nodes, relationships, and properties. They are particularly suitable for applications in which relationships and connections between entities are important. Neo4j is a popular graph database that uses Cypher as its declarative query language. Common applications include social networking, recommendation systems, fraud detection, knowledge graphs, and network management.

The major benefits of NoSQL databases include flexible data structures, scalability, large-scale data handling, efficient data access, distributed storage, and support for different data models. However, they also present challenges such as lack of a standardized query language, schema-related complexity, consistency considerations, limitations in handling certain complex relationships, and different data-modelling approaches.

Thus, NoSQL databases provide alternative approaches to data management and are particularly useful for modern applications where data volume, variety, scalability, flexibility, distributed processing, or complex relationships are important.

SGOU

Assignments

1. Explain the concept of NoSQL databases. Discuss their major characteristics, benefits, and challenges with suitable examples.

2. Explain the four major types of NoSQL databases, such as Key-Value Stores, Document Stores, Column-Family (Wide-Column) Stores, and Graph Databases with suitable examples.

3. Explain Key-Value Stores in detail. Describe their characteristics, applications, and working with a suitable example of Amazon DynamoDB.

4. Explain Document Stores in detail. Describe the structure and characteristics of document databases and explain MongoDB with a suitable example.

5. Explain Column-Family (Wide-Column) Stores in detail. Describe their characteristics and explain the structure of Apache Cassandra from cluster to columns with a suitable example.

6. Explain Graph Databases in detail. Describe nodes, relationships, and properties, and explain Neo4j and its applications with suitable examples.

Reference

1. Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed., McGraw-Hill, 2020

2. Elmasri, Ramez, and Shamkant B. Navathe. *Fundamentals of Database Systems*. 7th ed., Pearson, 2017.

3. García-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed., Pearson, 2008

Suggested Reading

1. Kaufmann, Michael, and Andreas Meier. *SQL and NoSQL Databases: Modeling, Languages, Security and Architectures for Big Data Management*. 2nd ed., Springer, 2023.

2. Aleksendrić, Marko, et al. *Mastering MongoDB 7.0: Achieve Data Excellence by Unlocking the Full Potential of MongoDB*. 4th ed., Packt Publishing, 2024.

3. Patil, Manisha M. *NoSQL Databases*. 1st ed., Iterative International Publishers, 2024

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

2 UNIT

Vector Databases

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the basic concepts, principles, and use cases of vector databases
- describe the working process of vector databases, including data chunking, embedding generation, indexing, and similarity search
- differentiate between distance metrics and vector indexing techniques used for efficient similarity search
- explain the applications and features of vector database providers such as Pinecone, Qdrant, Weaviate, and Milvus

SGOU

Background

The rapid growth of unstructured data such as text documents, images, audio, video, web content, and product information has created a need for search techniques that can understand meaning rather than only exact words. Traditional databases are mainly designed for structured data and exact-condition searches. Vector databases address this limitation by converting data into numerical vector representations called embeddings and retrieving information based on the similarity between vectors.

Vector databases have become important in modern Artificial Intelligence, semantic search, Retrieval-Augmented Generation (RAG), recommendation systems, question-answering systems, image search, and document retrieval. This unit introduces the concepts behind vector databases, their working principles, applications, embeddings, distance and similarity measures, indexing and querying techniques, challenges, and commonly used vector database providers.

Keywords

Vector Database, Vector, Embedding, Semantic Search, Similarity Search, Distance Metrics, Vector Indexing, Top-K Query

Discussion

4.2.1 Basics of Vector Database

A vector database is a specialized database designed to store, manage, and retrieve vector representations of data. In modern computing, a large amount of information is available in unstructured formats such as text documents, images, audio, video, product descriptions, and web content. Traditional databases are highly effective for structured information and exact searches, but they are not primarily designed to search information according to its semantic meaning. Vector databases address this requirement by representing data as numerical vectors and finding information based on the similarity between these vectors.

For example, consider the following two questions:

"How can I reset my password?"

and

"I forgot my password. How do I change it?"

The words are not exactly the same, but both questions express a similar meaning. A vector database can identify this relationship because an embedding model converts both sentences into numerical vectors, and the resulting vectors can be compared for similarity.

What is a Vector?

A vector is an ordered collection of numerical values. In a vector database, these values represent the characteristics or semantic information of an object.

For example:

V=[0.21, 0.45, 0.73, 0.18, 0.62]

The above vector is only an illustrative example. In real applications, vectors may contain hundreds or thousands of dimensions. A vector can represent text, an image,

SGOU

a document, a product, a user profile, or another type of data.

The important point is that the vector provides a mathematical representation that allows the system to compare different objects.

What is a Vector Database?

A vector database stores these numerical vector representations and provides mechanisms for searching vectors according to their similarity. Along with a vector, the database can store the original data or a reference to it and associated metadata.

For example:

Document: Introduction to Machine Learning
Vector: [0.21, 0.45, 0.73, ...]
Category: Artificial Intelligence
Year: 2026

When a user searches for information related to machine learning, the search query can also be converted into a vector. The vector database compares the query vector with stored vectors and retrieves the most relevant results.

Difference from Traditional Databases

A traditional database is generally designed to answer questions such as:

"Find the student whose Student_ID is 101."

This is an exact or structured search.

A vector database is designed for questions such as:

"Find documents related to machine learning applications."

The second query may not contain the exact title or wording of the relevant document. Instead, the system attempts to find documents that are semantically similar.

Thus, traditional databases primarily focus on structured data and exact conditions, whereas vector databases are particularly useful for similarity-based retrieval.

4.2.2 Principles of Vector Database

The operation of a vector database is based on several fundamental principles.

1. **Numerical Representation**

The first principle is to represent complex data in numerical form. Since computers can efficiently perform mathematical operations on numbers, text, images, and other objects are converted into vectors using embedding models.

SGOU

2. Semantic Representation

A vector should capture useful characteristics or meaning from the original data. When an appropriate embedding model is used, semantically related objects tend to produce similar vector representations.

For example:

"Car"

and

"Automobile"

have related meanings. Their embeddings are therefore expected to be relatively close in the vector space.

This principle allows vector databases to perform semantic search rather than relying only on exact words.

3. Similarity-Based Retrieval

The central principle of a vector database is similarity-based retrieval. A query is converted into a vector and compared with stored vectors. The database retrieves vectors that are closest or most similar to the query according to a selected similarity or distance measure.

For example:

Query: "Methods for protecting computer networks"

A document titled:

"Network Security Techniques"

may be retrieved because its vector representation is semantically related to the query.

4. Vector Space

Vectors can be considered as points in a mathematical space called a vector space. Similar objects are represented by vectors that are close to each other, while unrelated objects generally have less similar representations.

Real embeddings normally have many dimensions, so their vector spaces cannot easily be visualized.

5. Efficient Search

A vector database may contain millions or even billions of vectors. Comparing a query against every vector can be expensive. Therefore, vector databases use specialized indexing techniques to improve search efficiency.

Methods such as FLAT, IVF, PQ, Annoy, and HNSW are commonly associated with vector indexing and will be discussed separately.

SGOU

6. Metadata Filtering

Vector search can be combined with metadata conditions. For example, a user may request:

"Find machine learning documents published after 2025."

The system can perform semantic vector search while also applying a metadata condition such as:

Category = Machine Learning
Year > 2025

This combination helps improve the relevance of retrieved information.

4.2.3 Use Cases of Vector Database

Vector databases are widely used in AI and data-intensive applications where similarity-based retrieval is required.

1. Semantic Search

Semantic search retrieves information based on the meaning of a query rather than simply matching exact keywords.

For example, a learner searches:

"How can I recover access to my account?"

The system may retrieve:

"Account Password Reset Procedure"

even when the exact phrase "recover access" does not occur in the document.

This is useful for searching large collections of documents, websites, manuals, policies, and knowledge bases.

2. Retrieval-Augmented Generation

Vector databases are an important component of Retrieval-Augmented Generation (RAG) systems. In RAG, relevant information is retrieved from an external knowledge source and provided to a Generative AI model before generating an answer.

For example, an institution may store its regulations and academic documents in a vector database. When a learner asks:"What is the procedure for applying for a supplementary examination?"

the system can retrieve the relevant examination documents and provide them to the language model.

The basic process is:

SGOU

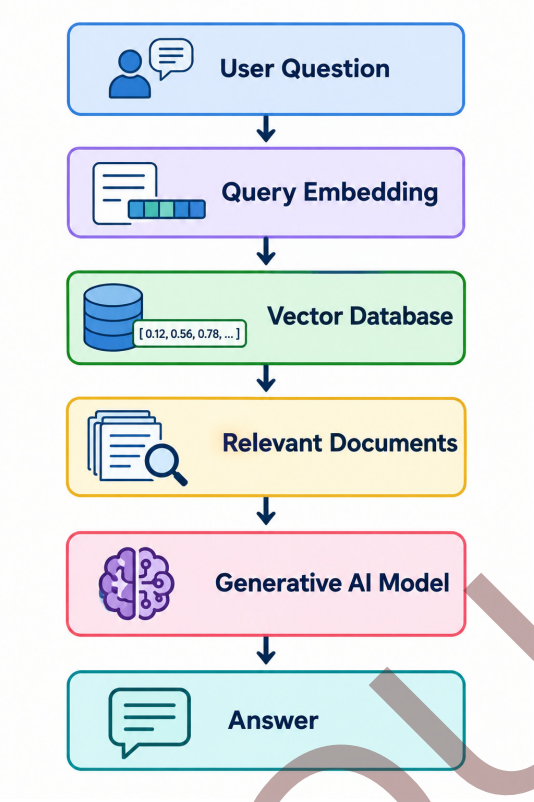


Fig 4.2.1 Process of RAG

3. Recommendation Systems

Vector databases can be used to recommend products, movies, books, music, or other items.

For example, if a customer shows interest in a particular laptop, the system can search for products with similar vector representations and recommend related laptops.

4. Image Search

Images can be converted into vector representations using image embedding models. The vector database can then retrieve visually or semantically similar images. This is useful in e-commerce, digital libraries, visual search, and image management systems.

5. Question-Answering Systems

Organizations can store FAQs, manuals, policies, guidelines, and other documents as vectors. When a user asks a question, the vector database can retrieve relevant sections.

For example:

Question: "What documents are required for admission?"

The system searches the institutional knowledge base and retrieves the relevant admission guidelines.

SGOU

6. Customer Support

Customer-support systems can use vector databases to store product manuals, FAQs, troubleshooting guides, and previous support information.

For example:

Customer: "My laptop is getting very hot while charging."

The system can retrieve troubleshooting information related to overheating and charging problems, even when the exact wording is different.

7. Document Retrieval

Organizations often maintain thousands of documents. Vector databases allow users to search such collections using natural-language questions rather than remembering exact document titles or keywords.

8. Fraud and Anomaly Detection

User behaviour or transactions can be represented as vectors. Similarity-based analysis can help identify unusual patterns. For example, a transaction that is substantially different from a user's normal behaviour may be identified for further investigation.

4.2.4 Working of a Vector Database

A vector database works by converting unstructured or semi-structured information such as text, images, audio, or documents into numerical vectors called embeddings. These vectors are then stored in a specialized database along with the original data and related metadata. When a user submits a query, the query is also converted into a vector. The vector database compares the query vector with stored vectors using a suitable similarity or distance measure and retrieves the most relevant results.

1. Data Collection

The first step is to collect the data that needs to be stored and searched. The data may include documents, web pages, product descriptions, customer queries, images, research papers, or other types of information. Large documents are often divided into smaller sections called chunks so that each part can be represented and searched more effectively.

Example: “A university wants to create a system that can answer questions from its study materials. The university collects PDF notes, textbooks, question papers, and other academic documents.”

2. Data Chunking

When the source data contains large documents, the documents are divided into smaller meaningful sections. This process is called chunking. Each chunk can then be converted into an embedding separately. Chunking helps the system retrieve only the relevant part of a document instead of returning the entire document.

SGOU

Example: "A 50-page Data Science textbook can be divided into smaller sections such as Introduction to Data Mining, Classification, Clustering, Association Rules, and so on. Each section can be stored as a separate chunk."

3. Embedding Generation

After the data is prepared, an embedding model converts each chunk or item into a numerical vector. The vector represents important characteristics or semantic meaning of the original information. Similar pieces of information are generally represented by vectors that are close to one another in the vector space.

For example, the sentence "Machine learning is used to analyse data" may be converted into a vector containing hundreds or thousands of numerical values.

Example: "Two sentences such as 'Machine learning is useful for data analysis' and 'ML can analyse large amounts of data' may have different words but similar meanings. Their embedding vectors are therefore expected to be relatively close."

4. Storing Vectors

The generated vectors are stored in the vector database. The database normally stores not only the vector but also the original content and metadata associated with it. Metadata may include document name, author, subject, date, category, or learner/programme information.

Table 4.2.1 Storing vector in vector database

Content	Vector	Metadata
Introduction to Machine Learning	[0.21, 0.45, 0.73, ...]	Subject: ML
Classification Techniques	[0.18, 0.52, 0.69, ...]	Subject: ML

5. Indexing the Vectors

When the number of stored vectors becomes very large, comparing a query with every vector can be time-consuming. Therefore, vector databases create indexes to make similarity searching faster.

Different indexing techniques such as FLAT, IVF, Product Quantization (PQ), Annoy, and HNSW can be used depending on the requirements of the application. These indexes organize the vectors in ways that allow the database to find likely nearest vectors efficiently.

Example: "If a database contains 10 million document vectors, checking all 10 million vectors for every question would require considerable computation. An index helps narrow down the search space and return relevant documents much faster."

SGOU

6. Query Processing

When a user enters a question or search phrase, the query is first processed by the same or a compatible embedding model used for the stored data. The query is converted into a query vector.

For example:

User Query: "What is machine learning?"

The embedding model converts this question into a numerical representation:

Query Vector: [0.24, 0.47, 0.71, ...]

The vector database then uses this query vector to search the stored vectors.

7. Similarity or Distance Calculation

The query vector is compared with the vectors stored in the database. The database uses a mathematical measure to determine how close or similar the vectors are.

Common measures include cosine similarity, Euclidean distance, and dot product. Depending on the selected measure, vectors with greater similarity or smaller distance are considered more relevant.

Example: "If the query vector is very close to the vector representing a document about Machine Learning, that document receives a high similarity score and is considered relevant to the query."

8. Nearest Neighbour Search

The vector database performs a nearest neighbour search to identify the vectors that are closest to the query vector. In many applications, the system retrieves the Top-K most relevant vectors rather than only one result.

For example, if the user asks a question and the system is configured to return the top 5 results, the database retrieves the five vectors with the highest relevance according to the selected search method.

Example: "A learner searches for 'methods used for classification'. The vector database may retrieve sections discussing Decision Trees, Naïve Bayes, Support Vector Machines, and Random Forests because their embeddings are semantically related to the query."

9. Metadata Filtering

Vector databases can also use metadata to restrict the search. This allows the system to combine semantic similarity search with conventional filtering.

Example: "A learner searches for 'machine learning algorithms' but wants results only from the BCA programme and Semester 4. The database can search for semantically similar vectors while filtering the results using metadata such as Programme = BCA and Semester = 4."

SGOU

This makes the search more precise and useful.

10. Retrieving the Original Information

After identifying the most relevant vectors, the database returns the associated original content and metadata. The application can then display the retrieved information to the user or pass it to another system.

In an AI question-answering system, the retrieved information may be supplied to a Large Language Model (LLM) as context. The LLM can then generate an answer based on the retrieved information. This approach is commonly used in Retrieval-Augmented Generation (RAG) systems.

4.2.5 Challenges of Vector Database

1. Storage Requirements

Vectors can contain hundreds or thousands of numerical dimensions. When millions of vectors are stored, the storage requirement can become substantial. The database must store vectors, indexes, metadata, and sometimes the original data or references to it.

2. Computational Requirements

Embedding generation and vector search require computational resources. Large datasets may require powerful CPUs, GPUs, memory, or distributed infrastructure depending on the workload.

3. Embedding Quality

The quality of the embedding directly affects search quality. If the embedding model does not capture the relevant meaning or characteristics of the data, the vector database may retrieve irrelevant information. For example, if two concepts are related in the real world but their embeddings do not represent that relationship effectively, a similarity search may fail to retrieve the expected result.

4. High-Dimensional Data

Modern embeddings can have a large number of dimensions. Searching high-dimensional data efficiently is challenging because conventional search techniques may become expensive as the number of dimensions increases.

This problem is commonly associated with the curse of dimensionality.

5. Retrieval Accuracy

A vector database retrieves information based on mathematical similarity. Similarity does not always mean factual correctness or usefulness. For example, two documents may use similar words but discuss different topics. Therefore, embedding quality, chunking strategy, similarity measures, and indexing methods all influence retrieval accuracy.

SGOU

6. Data Updating

When the original document changes, its embedding may also need to be regenerated. For example:

Maintaining up-to-date embeddings can create additional processing requirements.

7. Security and Privacy

Vector databases may contain sensitive business information, personal information, confidential documents, or proprietary knowledge. Therefore, appropriate authentication, authorization, encryption, and access-control mechanisms are necessary.

8. Scalability

As the number of vectors increases, maintaining low search latency becomes more difficult. Large-scale applications need suitable indexing, storage architecture, and distributed processing to maintain performance.

9. Cost

Large vector databases may require significant storage, memory, computing resources, and infrastructure. The overall cost increases with the number of vectors, frequency of queries, embedding generation, and required performance.

4.2.6 Embedding

Embedding is the process of converting data such as text, images, audio, or documents into a numerical representation called a vector. A vector is an ordered collection of numbers that represents important characteristics or semantic meaning of the original data. Embeddings allow a computer to represent information in a mathematical form so that it can compare different pieces of data based on their meaning.

In a vector database, embeddings are very important because the database does not directly compare long sentences or documents in their original form. Instead, it compares their numerical vector representations. An embedding model analyses the input data and generates a vector containing many numerical values.

Example: “The sentence ‘Machine learning is used for data analysis’ can be converted into an embedding such as [0.21, 0.45, 0.73, 0.18, ...]. The actual embedding may contain hundreds or thousands of numerical values.”

1. Semantic Representation

One of the important properties of an embedding is that it can represent the semantic meaning of data. Information with similar meanings tends to have vectors that are located close to each other in the vector space.

Example: “Machine learning is useful for analysing data” and “ML helps computers learn patterns from data” use different words, but their meanings are related. Their

SGOU

embedding vectors may therefore be close to each other."

This allows a vector database to perform semantic search, where results can be retrieved based on meaning rather than exact keyword matching.

2. **Query Embedding**

When a user enters a query, the query is also converted into an embedding using an embedding model. This query vector is then compared with the vectors already stored in the vector database.

Example: "If a learner enters the query 'methods for detecting network attacks', the system converts the query into a vector and searches for stored vectors representing information about intrusion detection, network attacks, and related techniques."

3. **Document Embedding**

Documents can be divided into smaller sections or chunks before generating embeddings. Each chunk receives its own vector. This allows the vector database to retrieve the specific section that is relevant to a query.

Example: "A 100-page cybersecurity textbook can be divided into sections such as DoS attacks, Brute Force attacks, Botnets, and Intrusion Detection Systems. Each section can be converted into a separate embedding and stored in the vector database."

4.2.7 Distance Metrics and Similarity

After converting data and queries into vectors, the vector database needs a mathematical method to determine how close or similar two vectors are. Distance metrics measure the distance between vectors, while similarity measures determine how similar the vectors are.

The choice of metric affects the results produced by a vector search system. Common measures include Euclidean distance, Cosine similarity, Dot Product, and Manhattan distance.

1. **Euclidean Distance**

Euclidean distance measures the straight-line distance between two points in a vector space. It is commonly used when the actual geometric distance between vectors is important.

For two vectors A and B, Euclidean distance is calculated as:

$$d(A,B) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$$

A smaller Euclidean distance means that the vectors are closer to each other.

SGOU

Example: “Suppose vector A is [2, 3] and vector B is [5, 7]. The Euclidean distance is

$$\sqrt{(2-5)^2+(3-7)^2} = \sqrt{9+16} = 5$$

In a vector database, the database can calculate the Euclidean distance between the query vector and stored vectors and retrieve the vectors with the smallest distances.

2. Cosine Similarity

Cosine similarity measures the similarity between two vectors by calculating the angle between them rather than simply measuring their physical distance. It is widely used for text and document embeddings.

The formula is:

$$Cosine\ Similarity(A,B) = \frac{A \cdot B}{||A||\ ||B||}$$

The value generally ranges from -1 to +1. A value closer to 1 indicates that the vectors have a similar direction and therefore are more similar.

Example: “The vectors representing ‘database management’ and ‘DBMS concepts’ may have a high cosine similarity because their semantic meanings are closely related.”

Cosine similarity is particularly useful when the direction of the vector is more important than its magnitude.

3. Dot Product

The dot product is another method used to compare vectors. It is calculated by multiplying corresponding values and adding the results.

$$A \cdot B = \sum_{i=1}^{n} A_i B_i$$

For example:

A=[2,3] B=[4,5]

Then:

A·B=(2×4)+(3×5)=8+15=23

A larger dot-product value can indicate greater similarity when the embedding/search setup is designed for dot-product comparison. Unlike cosine similarity, dot products can also be affected by the magnitude of the vectors.

Example: “In a recommendation system, a product vector and a user-interest vector can be compared using their dot product to identify products that closely match the user's represented interests.”

SGOU

4. Manhattan Distance

Manhattan distance measures the distance between two vectors by adding the absolute differences between their corresponding values.

The formula is :

$$d(A,B) = \sum_{i=1}^{n} |A_i - B_i|$$

For example:

A=[2,3] B=[5,7]

Therefore:

d(A,B)=|2-5|+|3-7| =3+4=7=3+4=7

A smaller Manhattan distance indicates that the vectors are closer.

> Example: "If two vectors represent two similar products and their Manhattan distance is small, the products may be considered similar according to the selected embedding and search method."

Difference Between Distance and Similarity

Distance and similarity express related ideas in different ways. A distance measure usually considers smaller values to indicate greater closeness, whereas a similarity measure usually considers larger values to indicate greater similarity.

Example: "If two documents have a Euclidean distance of 0.15 and another pair has a distance of 0.85, the first pair is closer according to Euclidean distance. In contrast, if two documents have cosine similarities of 0.95 and 0.60, the pair with 0.95 is more similar according to cosine similarity."

Therefore, the vector database uses the selected metric to rank the retrieved vectors and identify the most relevant information.

4.2.8 Indexing and Querying

1.Vector Indexing

Vector indexing is the process of organizing vectors in a vector database so that similar vectors can be found quickly during a search. A vector database may contain thousands, millions, or even billions of vectors. Comparing a query vector with every stored vector can require a large amount of time and computational resources. Indexing reduces the amount of searching required by organizing vectors in a suitable structure.

Example: "Suppose a university has stored 10 million document embeddings in a vector database. If a learner asks a question, comparing the question vector with all 10 million vectors would be expensive. A vector index helps the database identify a smaller set of

SGOU

likely relevant vectors and search them efficiently."

Different indexing techniques provide different balances between search speed, accuracy, memory usage, and index-building cost. Important techniques include FLAT, IVF, PQ, Annoy, and HNSW.

4.2.8.1 FLAT Index

FLAT is the simplest vector indexing method. It performs an exact or exhaustive search by comparing the query vector with every vector stored in the database. No approximate structure is used to reduce the search space.

For example, if a database contains 1,000 vectors, a query vector is compared with all 1,000 vectors. The distance or similarity score is calculated for every vector, and the results are ranked to select the nearest vectors.

Example: "A college stores 5,000 student-document embeddings. When a learner searches for 'machine learning algorithms', the FLAT index compares the query embedding with all 5,000 embeddings and returns the most similar documents."

The major advantage of FLAT indexing is high accuracy, because every stored vector is considered. However, its main limitation is search speed. As the number of vectors increases, exhaustive comparison becomes computationally expensive.

FLAT indexing is therefore useful for small datasets, testing, benchmarking, and applications where exact nearest-neighbour results are important.

4.2.8.2 IVF (Inverted File Index)

IVF (Inverted File Index) improves search speed by dividing vectors into groups or clusters. A clustering algorithm, commonly based on k-means, creates a number of

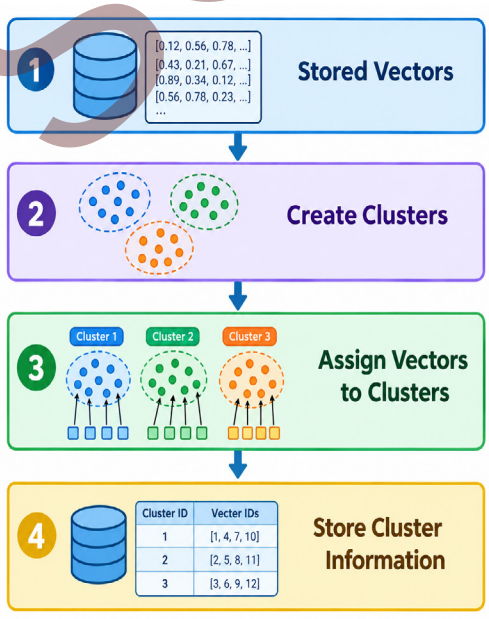


Fig 4.2.2 Process of IVF

SGOU

cluster centres. Each stored vector is assigned to the closest cluster.

Instead of searching the entire database for every query, IVF first determines which clusters are closest to the query vector. It then searches only those selected clusters.

The basic process is:

During a query:

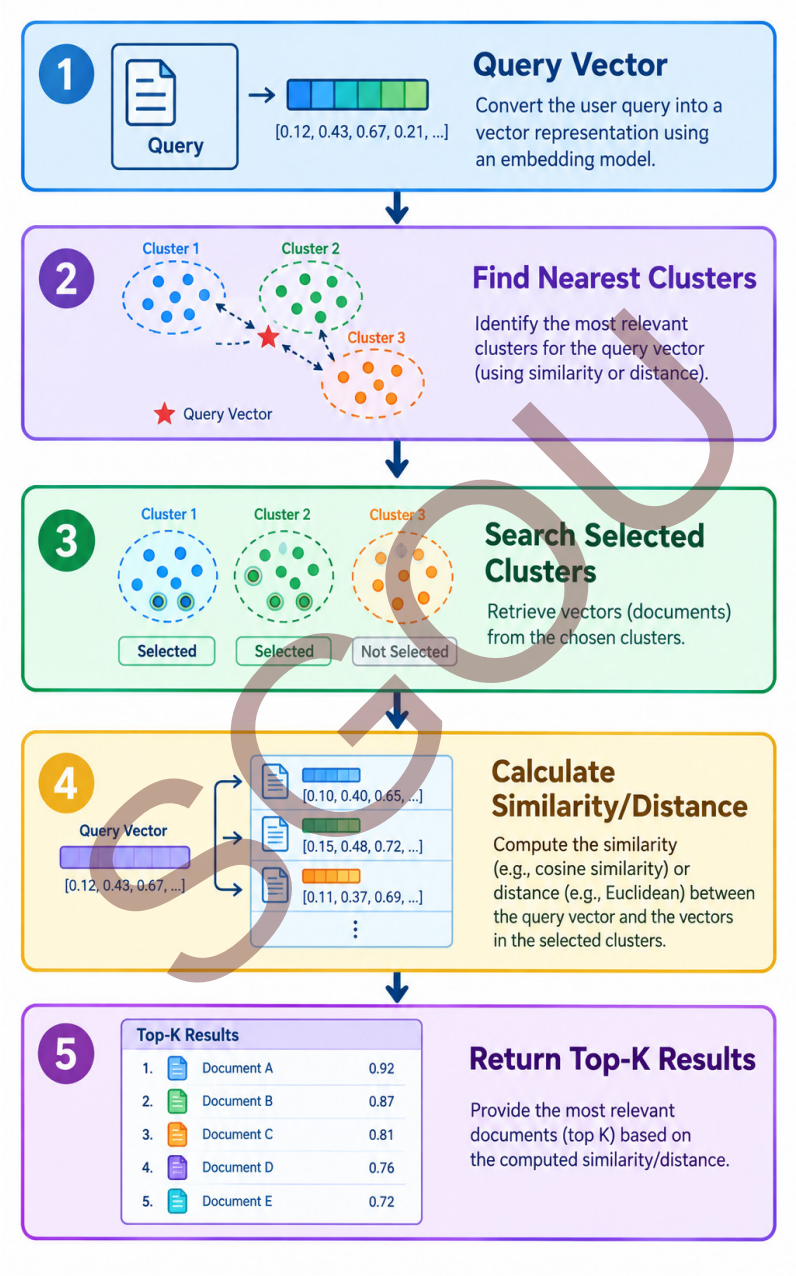


Fig 4.2.3 IVF Query process

Example: "Suppose 1 million document vectors are divided into 1,000 clusters. When a learner enters a query, the system may identify the 5 most relevant clusters and search mainly within those clusters instead of comparing the query with all 1 million vectors."

The number of clusters and the number of clusters searched during a query affect the balance between speed and recall. Searching more clusters generally increases the chance of finding the exact relevant vectors but also increases computation.

4.2.8.3 PQ (Product Quantization)

Product Quantization (PQ) is mainly used to reduce the memory required to store vectors and to make similarity search more efficient. Instead of storing every vector using its complete numerical representation, PQ divides a vector into smaller parts and represents each part using a compact code.

For example, a vector with many dimensions can be divided into several smaller sub-vectors. Each sub-vector is then assigned a short code representing its approximate value.

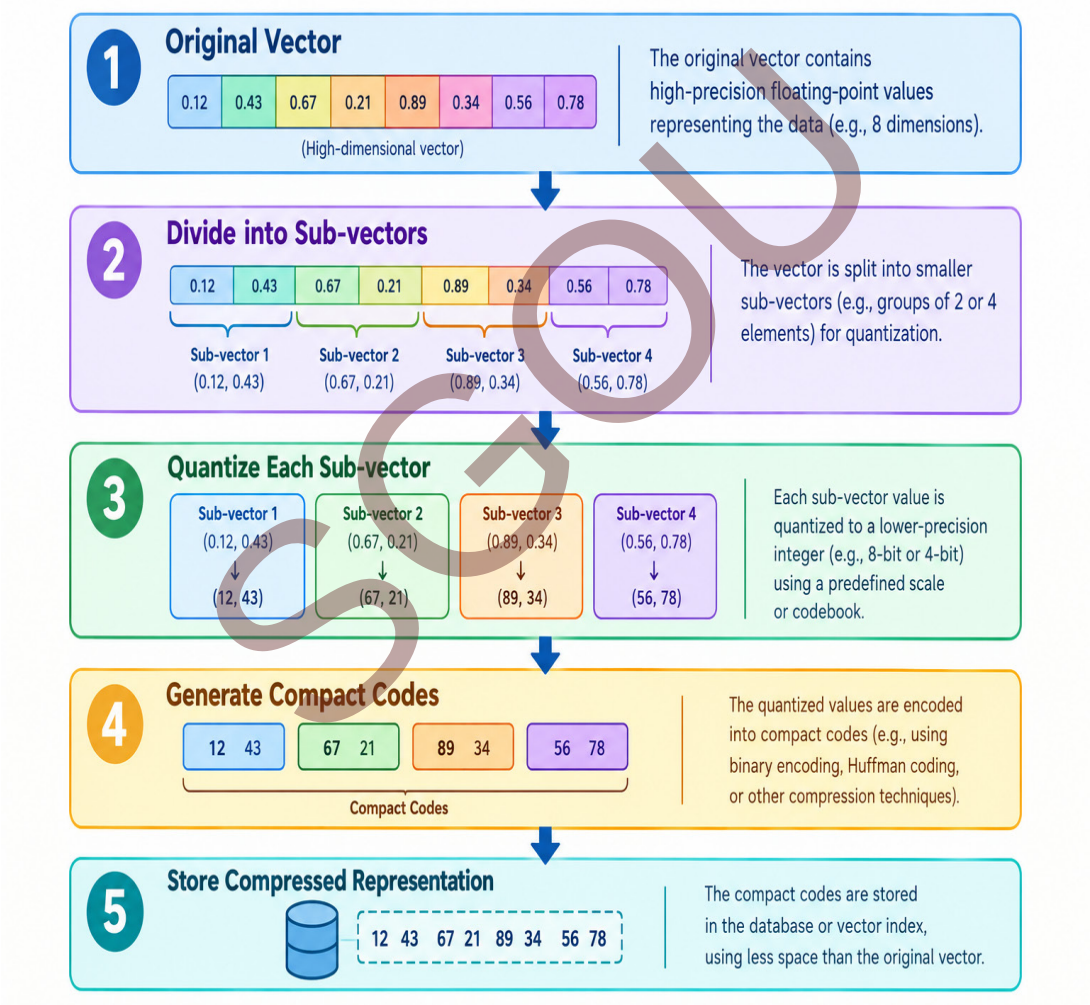


Fig 4.2.4 Product Quantization (PQ) Process

Example: "Suppose each document embedding requires a large amount of memory. PQ can compress the representation so that many more document vectors can be stored in memory while still allowing approximate similarity searches."

The major benefit of PQ is memory efficiency. It is particularly useful for very large datasets. However, because the vectors are represented approximately rather than exactly, some accuracy can be lost.

PQ is also commonly combined with IVF, producing an IVF-PQ approach. IVF reduces the search space, while PQ reduces the memory and computational requirements of vector representations.

4.2.8.4 Annoy(Approximate Nearest Neighbors Oh Yeah)

Annoy, whose name comes from Approximate Nearest Neighbors Oh Yeah, is an approximate nearest-neighbour search technique based on trees. It creates multiple tree structures that divide the vector space into regions.

During a search, the query vector is passed through these trees to identify regions that are likely to contain nearby vectors. Instead of checking every stored vector, the system searches selected candidate vectors.

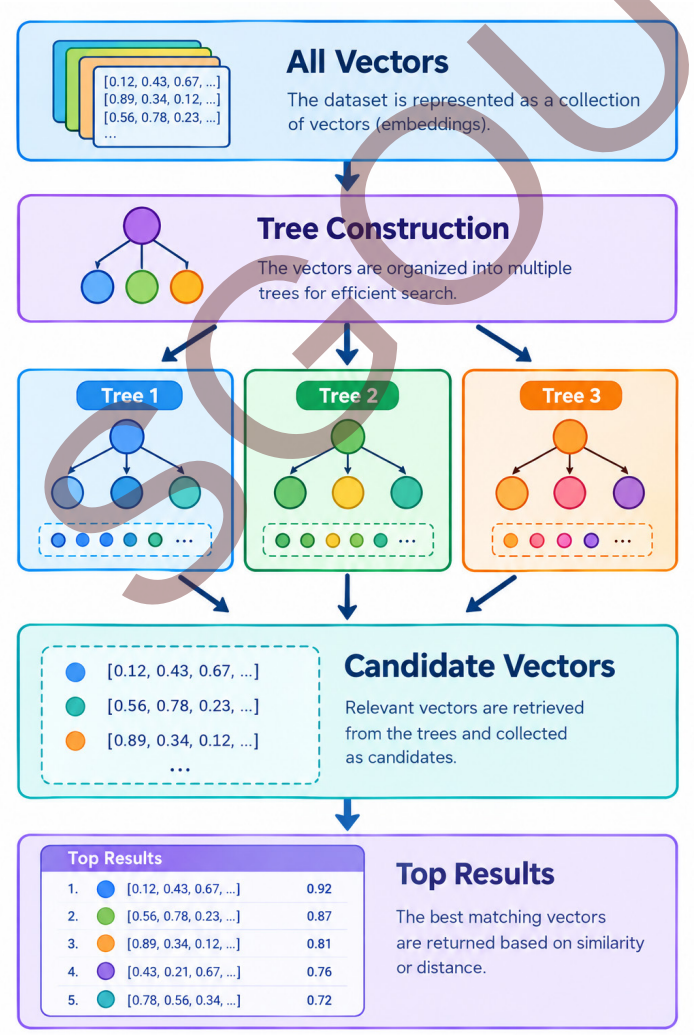


Fig 4.2.5 Annoy(Approximate Nearest Neighbors Oh Yeah) process

Example: “Suppose a recommendation system stores millions of product embeddings. Annoy can construct multiple trees that organize these products into regions. When a customer searches for a product, the system explores the relevant regions instead of comparing the query with every product.”

Annoy can provide fast approximate search and is useful when the dataset is relatively stable. Its tree-based structure can be efficient for read-heavy applications, although frequent updates may require rebuilding or managing the index differently.

4.2.8.5 HNSW (Hierarchical Navigable Small World)

HNSW (Hierarchical Navigable Small World) is a graph-based approximate nearest-neighbour indexing technique. It organizes vectors into multiple layers of a graph. The upper layers provide a coarse representation of the vector space, while the lower layers contain more detailed connections between vectors.

A simplified structure is:

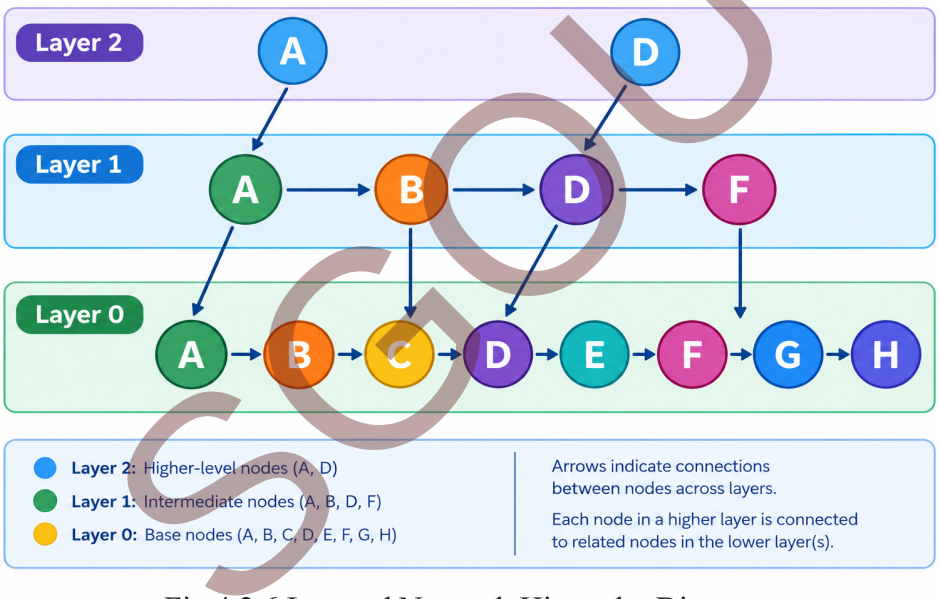


Fig 4.2.6 Layered Network Hierarchy Diagram

When a query is received, the search starts from a higher layer and moves toward a region that appears close to the query. It then moves down through the layers until it reaches the bottom layer, where a more detailed search is performed.

Example: “Imagine a map containing millions of locations. Instead of checking every location, HNSW first uses broad connections to move quickly toward the correct region and then uses more detailed connections to identify the nearest locations.”

HNSW is widely used because it can provide low search latency and high recall. Its main limitation is that graph structures can require considerable memory, and constructing the index can be computationally expensive for very large datasets.

Table 4.2.2 Comparison of Vector Indexing Methods

Indexing Method	Basic Approach	Search Type	Main Advantage	Main Limitation
FLAT	Compare with every vector	Exact	High accuracy	Slow for large datasets
IVF	Search selected clusters	Approximate	Reduces search space	May miss some results
PQ	Compress vector representations	Approximate	Saves memory	Some accuracy loss
Annoy	Multiple tree structures	Approximate	Fast search	Less suitable for frequent updates
HNSW	Multi-layer graph	Approximate	Fast and high recall	Higher memory usage

The methods can also be combined. For example, IVF-PQ uses IVF to restrict the search to relevant clusters and PQ to compress vector representations. Such combinations are useful when dealing with very large vector collections.

2. Vector Querying

Vector querying is the process of searching a vector database using a query vector to find the most relevant stored vectors. Unlike traditional database queries that generally search for exact values or keywords, vector queries search according to similarity or distance.

The general process is:

Example: "A learner enters the query 'Explain different types of machine learning'. The system converts this question into an embedding vector and submits it to the vector database. The database searches its index and identifies the document chunks whose vectors are most similar to the query vector."

Query Embedding

The first stage of vector querying is to convert the user's query into a numerical vector. The same or a compatible embedding model used during data preparation is normally used to generate the query embedding.

Example: "The query 'What are supervised learning algorithms?' is converted into a numerical vector such as [0.23, 0.51, 0.72, ...]. This vector is then used to search the vector database."

SGOU

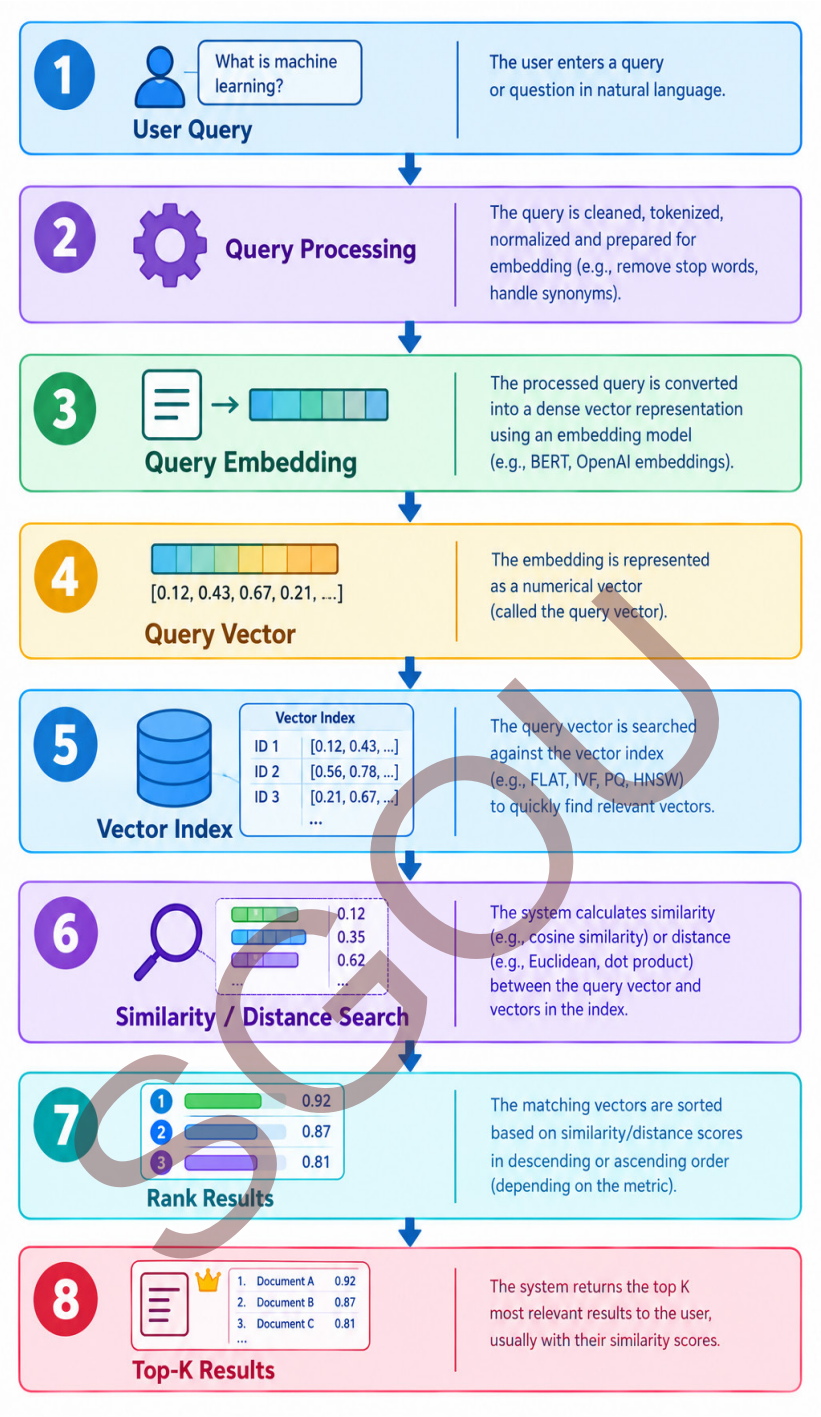


Fig 4.2.7 Vector querying process

The quality of the query embedding is important because it determines how effectively the system can identify semantically related information.

Similarity Search

After generating the query vector, the vector database compares it with stored vectors using a selected distance or similarity metric. Common measures include cosine similarity, Euclidean distance, and dot product.

For similarity measures, a higher score generally indicates greater similarity. For distance measures, a smaller value generally indicates that the vectors are closer.

Example: "If a query has a cosine similarity of 0.92 with one document and 0.58 with another document, the first document would normally be considered more semantically similar."

Top-K Query

A vector query often asks the database to return the Top-K most relevant results. Here, K represents the number of results required.

Example: "If K = 5, the vector database returns the five highest-ranked document chunks according to the selected similarity or distance measure."

Top-K retrieval is especially useful in recommendation systems, semantic search, and Retrieval-Augmented Generation (RAG) applications.

Metadata Filtering with Vector Querying

Vector querying can also be combined with metadata filters. This allows the system to search for semantically similar vectors while restricting the results according to specific conditions.

Example: "A learner searches for 'database normalization' and wants results only from Semester 4 BCA materials. The vector database can perform similarity search while applying metadata filters such as Programme = BCA and Semester = 4."

This combination makes vector search more precise and useful in practical applications.

4.2.9 Vector DB Providers

A Vector Database Provider is a company, platform, or software project that provides the infrastructure and tools required to store, index, manage, and search vector embeddings. Vector database providers make it easier for developers to build applications such as semantic search, recommendation systems, Retrieval-Augmented Generation (RAG), question-answering systems, and AI agents.

Instead of developing a complete vector-search system from the beginning, developers can use an existing vector database provider. The provider generally offers features for storing vectors, creating indexes, performing similarity searches, filtering using metadata, and connecting the database with AI and machine-learning applications.

Example: "A university wants to build an AI chatbot that answers questions from its study materials. Instead of developing its own vector storage and similarity-search system, the university can store document embeddings in a vector database such as Pinecone, Qdrant, Weaviate, or Milvus."

SGOU

4.2.9.1 Pinecone

Pinecone is a managed vector database designed for AI applications. Its current platform provides vector storage and search through a managed service, with capabilities including dense and sparse vectors, metadata filtering, namespaces, and real-time indexing. Pinecone's current serverless architecture separates storage from compute, allowing the service to scale without users having to manage the underlying server infrastructure.

Pinecone is particularly useful when developers want to build AI applications without managing the infrastructure required for a vector database. It can be used for semantic

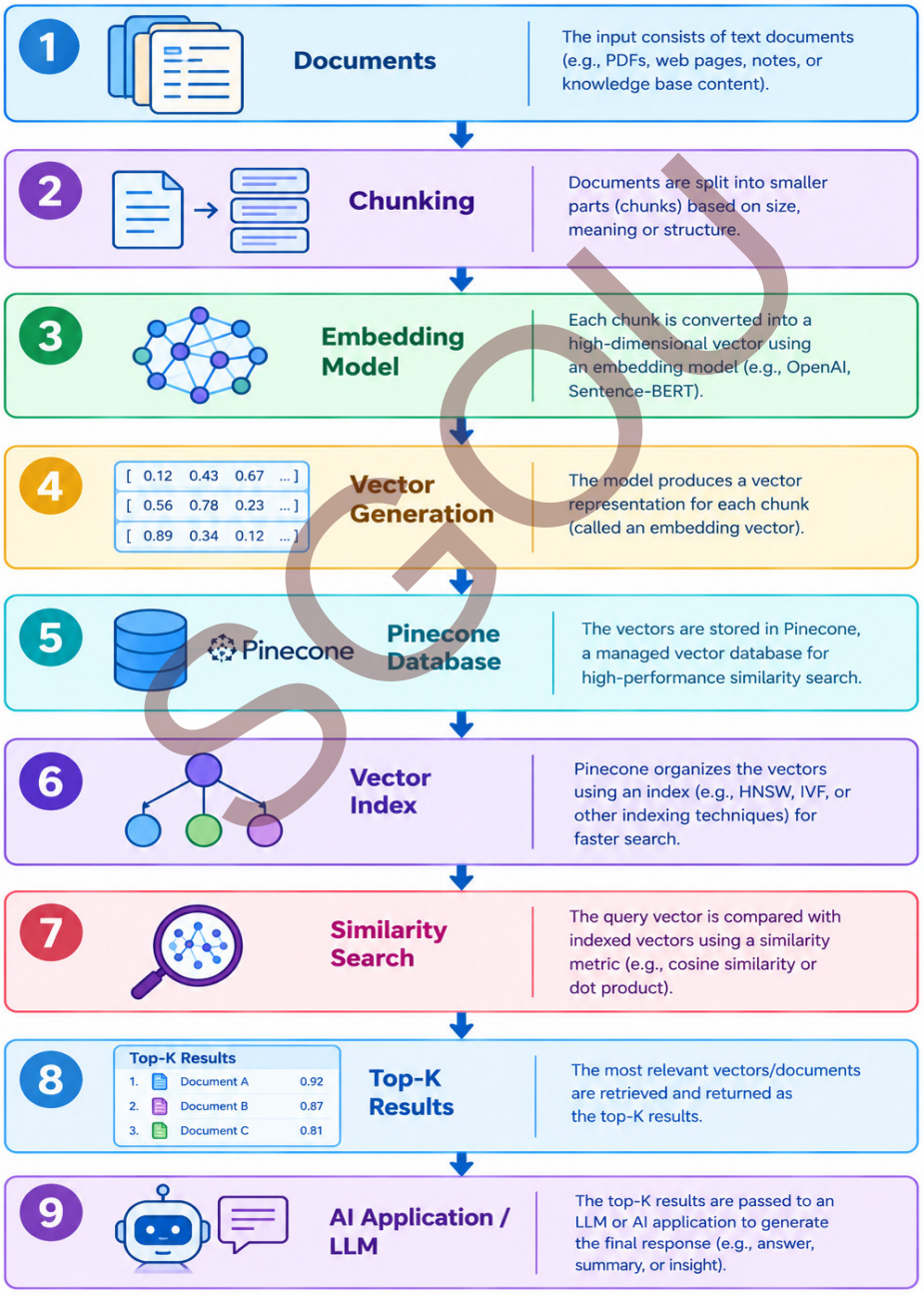


Fig 4.2.8 Pinecone workflow

SGOU

search, recommendation systems, RAG applications, and AI agents. Pinecone also supports both dense and sparse representations, allowing semantic and keyword-oriented retrieval strategies to be used within the platform.

Example: "An online education platform stores embeddings of thousands of course materials in Pinecone. When a learner asks, 'Explain supervised learning', the application creates a query embedding and sends it to Pinecone. Pinecone searches the stored vectors and returns the most relevant course-material sections."

Working with Pinecone

A simplified Pinecone workflow is:

Pinecone is a managed service, so the developer does not need to manually maintain the complete database infrastructure. This makes it suitable for applications where simplified deployment and cloud scalability are important.

Pinecone is a managed service, so the developer does not need to manually maintain the complete database infrastructure. This makes it suitable for applications where simplified deployment and cloud scalability are important.

4.2.9.2 Qdrant

Qdrant is a vector search engine and vector database designed for similarity search and AI retrieval applications. It supports vector search, metadata filtering, and hybrid search involving dense and sparse representations. Qdrant can be deployed using different deployment models, including its cloud offering and self-managed deployments.

Qdrant stores vectors together with associated information, often called payload. Payload information can be used to filter search results. This is useful when semantic similarity alone is not sufficient.

Example: "An online shopping application stores product-image or product-description embeddings in Qdrant. A customer searches for 'comfortable running shoes'. The system can search for semantically similar products while also filtering the results according to conditions such as price, availability, or product category."

Qdrant also provides hybrid search, where semantic vector search can be combined with lexical or keyword-based search. This is useful when an application needs both meaning-based matching and exact-term matching.

4.2.9.3 Weaviate

Weaviate is an open-source vector database designed for AI applications. It can store objects together with their vector embeddings and supports semantic and hybrid search. Weaviate can be deployed as a managed cloud service or self-hosted, including deployment with Docker or Kubernetes.

Weaviate provides vector search as well as hybrid search, which combines vector-based semantic search with keyword search. It also provides filtering, vector index options, multi-tenancy, and integrations with different AI models.

SGOU

Example: “A university can store course documents in Weaviate and search them using both semantic meaning and exact keywords. A query such as ‘machine learning’ can retrieve documents that discuss the concept even when the exact wording differs, while keyword search can help find documents containing a specific course code.”

4.2.9.4 Milvus

Milvus is a vector database designed for large-scale vector data and similarity search. Its documentation provides capabilities for managing collections, inserting and deleting data, vector search, hybrid search, metadata queries, indexing, and distributed deployment.

Milvus is designed for applications that may need to handle very large collections of vectors. Its platform supports distributed scaling, metadata filtering, hybrid search, and multiple vector-related capabilities.

Example: “A large e-commerce company may store millions or billions of product embeddings in Milvus. When a customer searches for a product, Milvus can perform a vector similarity search and return products that are semantically similar to the query.”

Summarised Overview

This unit explains how vector databases represent information as numerical vectors and use similarity-based retrieval to find semantically related information. It covers the basic principles of vector databases, their major applications, and the complete working process from data collection and chunking to embedding generation, vector storage, indexing, query processing, similarity calculation, and retrieval.

The unit also discusses embeddings and explains how they represent the semantic meaning of data. Different distance and similarity metrics, including Euclidean distance, Cosine similarity, Dot Product, and Manhattan distance, are introduced. The unit further explains important vector indexing methods such as FLAT, IVF, PQ, Annoy, and HNSW, followed by vector querying and Top-K retrieval. Finally, it introduces major vector database providers such as Pinecone, Qdrant, Weaviate, and Milvus, showing their role in developing modern AI applications.

SGOU

Assignments

1. Explain the basic concepts and principles of vector databases.
2. Explain the working of a vector database from data collection to retrieval of the original information.
3. Explain the concept of embeddings and discuss their role in vector databases.
4. Explain Euclidean distance, Cosine similarity, Dot Product, and Manhattan distance used in vector databases.
5. Explain FLAT, IVF, Product Quantization, Annoy, and HNSW indexing techniques.
6. Discuss the major use cases of vector databases in AI applications.
7. Explain the features and applications of Pinecone, Qdrant, Weaviate, and Milvus.

Reference

1. Han, J., Kamber, M., & Pei, J. (2011). *Data mining: Concepts and techniques* (3rd ed.). Morgan Kaufmann.
2. Jurafsky, D., & Martin, J. H. (2023). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition* (3rd ed., draft). Stanford University.
3. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.
4. Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). *Mining of massive datasets* (3rd ed.). Cambridge University Press.

SGOU

Suggested Reading

1. Pinecone. Vector database and vector search concepts. Pinecone Documentation.

2. Qdrant. Qdrant documentation: Vector search and retrieval. Qdrant Documentation.

3. Weaviate. Weaviate documentation: Vector database and semantic search. Weaviate Documentation.

4. Milvus. Milvus documentation: Vector database and similarity search. Milvus Documentation.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

3
UNIT

Cloud Databases

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the fundamental concepts of cloud databases
- describe the characteristics and types of cloud databases
- differentiate between managed and self-managed cloud databases
- analyse the use cases and challenges of cloud databases
- evaluate database services provided by AWS and IBM Cloud

Background

In traditional database systems, organisations need to purchase and maintain servers, storage devices, database software, networking equipment, and other infrastructure. The organisation is responsible for installing the database, performing backups, applying updates, monitoring performance, and ensuring availability. As applications increasingly require flexible storage, scalability, and access from different locations, managing all these resources locally can become difficult and expensive.

Cloud computing provides an alternative approach in which computing, storage, and database resources can be provided through the Internet. A cloud database allows organisations to store, manage, and access data using cloud-based infrastructure. Depending on the service model, the database infrastructure may be managed by the cloud provider, the organisation, or both.

SGOU

For example, an online shopping application may use a cloud database to store customer details, product information, orders, and payment-related records. Cloud databases can provide flexibility and allow organisations to scale resources according to application requirements.

In this unit, you will learn the fundamentals of cloud databases, their characteristics and types, and the differences between managed and self-managed cloud databases. You will also explore their use cases and challenges and examine database services provided by AWS and IBM Cloud.

Keywords

Cloud Database, Database as a Service (DBaaS), Cloud Computing, Managed Database, Self-Managed Database, Relational Database, NoSQL Database, Scalability, Availability, Data Security, AWS, Amazon RDS, Amazon DynamoDB, IBM Cloud

Discussion

4.3.1 Introduction

A cloud database is a database that is hosted and accessed through a cloud computing environment. Instead of maintaining all database servers and infrastructure within an organisation, users can utilize computing, storage, networking, and database resources provided through a cloud service provider.

Cloud databases have become important because modern applications often need to handle large amounts of data, varying workloads, and users from different locations. Cloud environments can provide resources that can be increased or decreased according to application requirements.

For example, an online shopping application may have a normal number of users on ordinary days but may experience a very high number of users and requests during a major sale. A cloud database can be configured to provide additional resources or scaling capabilities to handle increased workloads.

Consider an e-commerce application. The application may store information about customers, products, orders, payments, and deliveries. If the application is deployed in

SGOU

the cloud, its database can also be deployed using a cloud database service or on cloud-based infrastructure. The organisation can access the database through its applications, while the cloud platform can provide infrastructure and database-related services according to the selected service model.

Depending on the type of cloud database, the **cloud provider or the organisation** may be responsible for tasks such as database installation, configuration, maintenance, backups, updates, security, and performance monitoring.

Cloud databases are commonly used in:

- **E-Commerce Applications** – Managing customers, products, orders, and transactions.
- **Banking and Financial Applications** – Managing accounts, transactions, and financial records.
- **Healthcare Systems** – Managing patient, appointment, and medical information.
- **Educational Platforms** – Managing learners, courses, assessments, and academic records.
- **Social Media Applications** – Managing user profiles, posts, interactions, and activity data.
- **Mobile Applications** – Storing user and application data.
- **Business Information Systems** – Managing organisational and operational data.
- **Data Analytics Applications** – Storing and processing data for analysis and reporting.
- **Internet of Things (IoT) Applications** – Managing data generated by connected devices and sensors.

Depending on the type of cloud database and service model, responsibility for tasks such as database installation, configuration, maintenance, backups, updates, security, and performance monitoring may be shared between the cloud provider and the organisation.

4.3.2 Fundamentals of Cloud Databases

A cloud database combines the concepts of database management and cloud computing. It provides database storage and processing capabilities using cloud-based computing resources.

A traditional database normally requires an organisation to purchase, install, and maintain resources such as:

SGOU

- Database Servers
- Storage Devices
- Networking Equipment
- Backup Systems
- Security Infrastructure
- Database Software

In a cloud environment, many of these resources can be provided through a cloud service provider. Depending on the selected service model, some management responsibilities may remain with the organisation.

Basic Working of a Cloud Database

A simplified cloud database environment can be represented as:

Application → Internet/Private Network → Cloud Database Service → Cloud Infrastructure

The application sends database requests to the cloud database service. The cloud database service processes the requests using the underlying cloud resources and returns the required results to the application.

Example

Suppose a college develops an online examination system.

The system may store: the following information:

Table 4.3.1: Databases

Data	Example
Learner information	Learner ID, Name, Programme
Examination information	Course, Date, Time
Marks	Internal and External Marks
Results	Grade, Percentage
Login information	Username and account details

Instead of maintaining all database infrastructure on the college premises, the college can deploy the database using a cloud database service.

Cloud databases have several important characteristics.

1. **Scalability**

Cloud databases can be scaled to provide additional computing, storage, or other resources when application requirements increase.

SGOU

For example, an online shopping website may require additional database resources during a festival sale when the number of users and transactions increases.

2. **Accessibility**

Cloud databases can generally be accessed by applications from different locations through network connections, subject to appropriate authentication and security controls.

For example, a college application can access its cloud database from different campuses through a secure network connection.

3. **Elasticity**

Cloud resources can be dynamically increased or decreased according to changing workloads.

For example, additional database resources may be provided during periods of high activity and reduced when the workload decreases.

4. **Availability**

Cloud database services can provide mechanisms to improve availability and fault tolerance, such as redundancy, replication, backups, and failover mechanisms.

These capabilities depend on the particular cloud service and its configuration.

5. **Backup and Recovery**

Many cloud database services provide backup and recovery features, including automated backups and point-in-time recovery options.

These features help organizations recover data in the event of accidental deletion, system failure, or other problems.

6. **Security**

Cloud database platforms provide security mechanisms such as:

- Authentication
- Access control
- Encryption
- Network security controls
- Monitoring and logging

Security responsibilities are generally shared between the cloud provider and the organisation, depending on the selected service model.

SGOU

7. **Usage-Based Pricing**

Many cloud database services use usage-based or resource-based pricing models. Organisations may be charged according to factors such as computing resources, storage, database operations, data transfer, or selected service features.

This allows organisations to pay for the cloud resources and services they use rather than purchasing all infrastructure in advance.

4.3.3 Types of Cloud Databases

Cloud databases can be classified in different ways based on their database model and deployment or management approach.

Based on the database model, the major types include:

1. **Relational Cloud Databases**
2. **NoSQL Cloud Databases**

4.3.3.1 Relational Cloud Databases

A Relational Cloud Database is a cloud-based database that stores data in tables consisting of rows and columns. It generally uses SQL (Structured Query Language) to create, manage, retrieve, and manipulate data.

Relational cloud databases are suitable for applications where data has a well-defined structure and relationships between entities are important.

Example

Consider a learner database:

Table 4.3.2: learner database

Learner_ID	Name	Course
101	Anu	MCA
102	Rahul	MCA
103	Meera	MCA

Here:

- Learner_ID identifies each learner.
- Name stores the learner's name.
- Course stores the learner's course.
- Each row represents a learner record.
- Each column represents a particular attribute.

SGOU

Relationships between these tables can be represented using keys such as primary keys and foreign keys.

4.3.3.2 NoSQL Cloud Databases

NoSQL cloud databases are cloud-based database systems designed to store and manage data using models other than the traditional relational table structure. They are particularly useful for applications that handle large volumes of data, flexible or changing data structures, and distributed workloads.

Unlike relational databases, NoSQL databases do not necessarily require data to be organized into fixed tables with predefined columns.

Common NoSQL Models

NoSQL cloud databases can use different data models, including:

- **Key-Value Databases** – Store data as key-value pairs.
- **Document Databases** – Store data as documents, commonly using JSON-like structures.
- **Column-Family Databases** – Organize data into partitions, rows, and columns for distributed processing.
- **Graph Databases** – Store data using nodes, relationships, and properties.

Example

A document-based cloud database may store learner information as:

```
{
 "Learner_ID": 101,
 "Name": "Anu",
 "Course": "MCA"
}
```

Examples of NoSQL Cloud Services

Examples of cloud-based NoSQL database services include:

- **Amazon DynamoDB** – Supports key-value and document data models.
- **Amazon Neptune** – A managed graph database service.
- **Amazon Keyspaces** – A managed wide-column database service compatible with Apache Cassandra.

SGOU

Applications of NoSQL Cloud Databases

NoSQL cloud databases are commonly used in:

- **Social Media Applications** – Managing user profiles, posts, and interactions.
- **Real-Time Applications** – Handling frequently changing data and high request volumes.
- **Content Management Systems** – Storing flexible and semi-structured content.
- **IoT Applications** – Managing large volumes of device and sensor data.
- **Large-Scale Web Applications** – Supporting applications with large numbers of users and varying workloads.
- **Mobile Applications** – Storing user profiles, application data, and activity information.

4.3.4 Database Management Models

- Cloud platforms generally provide two common approaches for running and managing database workloads:
- Managed Cloud Databases (Database-as-a-Service / DBaaS)
- Self-Managed Cloud Databases (IaaS-Based)
- The main difference is who is responsible for managing the database software and underlying infrastructure.

4.3.4.1 Managed Cloud Databases

A managed cloud database is a database service in which the cloud provider manages many of the underlying infrastructure and database administration tasks.

1. Depending on the service, the cloud provider may manage activities such as:

- Hardware and resource provisioning
- Database software installation
- Software updates and patching
- Backup management
- Monitoring
- Availability and failover mechanisms
- Infrastructure maintenance

SGOU

2. The organisation mainly focuses on using and configuring the database, managing users and data, and supporting application requirements.

3. The exact division of responsibilities depends on the cloud service and its configuration.

Example

Suppose an organisation needs a relational database for an online learning platform.

Instead of purchasing a physical server and installing database software manually, the organisation can select a managed database service from a cloud provider.

The cloud provider manages much of the underlying infrastructure and routine database operations, while the organisation manages aspects such as:

- Database configuration
- Users and access permissions
- Application data
- Database-related application requirements

Examples

- **Amazon RDS**
- **Amazon Aurora**
- **IBM Cloud Databases for PostgreSQL**

Advantages of Managed Cloud Databases

1. **Reduced Administrative Work**

The organisation does not need to manage all the underlying infrastructure and routine database maintenance tasks manually.

2. **Easy Provisioning**

Database resources can generally be created and configured more quickly than setting up physical database infrastructure.

3. **Automated Maintenance**

Many managed database services provide automated or simplified maintenance activities such as software patching and updates.

4. **Backup Support**

Managed database services commonly provide backup and recovery features, although

SGOU

the available options depend on the selected service and configuration.

5. **Scalability**

Many managed database services provide options to increase or decrease resources according to workload and service capabilities.

6. **High Availability**

Managed database services may provide features such as **replication, redundancy, and failover** to support database availability. The exact availability features depend on the service and configuration.

4.3.4.2 Self-Managed Cloud Databases

A self-managed cloud database is a database deployed on cloud infrastructure in which the organisation is responsible for most of the database administration, configuration, maintenance, and operational tasks.

- The cloud provider supplies infrastructure resources such as:
- Virtual Machines
- Storage
- Networking
- Computing Resources

The organisation is responsible for installing and managing the operating system and database software and for performing many database administration tasks.

Example

An organisation can create a virtual machine on Amazon EC2 and manually install MySQL, PostgreSQL, or MongoDB on it.

The organisation may then be responsible for:

- Installing the Database Software
- Configuring the Database
- Applying Updates and Patches
- Managing Users and Access Permissions
- Managing Backups and Recovery
- Monitoring Database Performance
- Managing Database Security

SGOU

- Troubleshooting Database Problems
- Maintaining the Operating System and Database Environment

Example Services

Self-managed databases can be deployed by installing database software manually on cloud infrastructure, such as:

- PostgreSQL on Amazon EC2
- MySQL on Amazon EC2
- MongoDB on Amazon EC2

Advantages of Self-Managed Cloud Databases

1. **Greater Control**

The organisation has greater control over the database environment, configuration, and administrative settings.

2. **Customisation**

Database software and the supporting environment can be configured according to specific application requirements.

3. **Flexibility**

Organisations can select specific software versions, extensions, configurations, operating-system settings, and supporting tools according to their needs.

4. **Suitable for Special Requirements**

Self-managed databases may be useful when an application requires configurations, database versions, extensions, or administrative controls that are not available through a particular managed database service.

5. **Greater Administrative Control**

The organisation can directly manage database maintenance, tuning, security configurations, and other operational activities.

SGOU

4.3.5 Managed vs Self-Managed Cloud Databases

Table 4.3.1: Comparison of Managed and Self-Managed Cloud Databases

Feature	Managed Cloud Database	Self-Managed Cloud Database
Infrastructure management	Mostly handled by provider	Provided by cloud provider
Database installation	Generally handled by provider	Organisation manages
Updates & Patches	Provider handles many routine updates	Organisation manages
Configuration Control	Limited to service capabilities	Greater control
Customisation	Depends on service	HGenerally higher
Administrative effort	Lower	Higher

4.3.6 Use Cases of Cloud Databases

Cloud databases are used in a wide range of applications because they can provide scalability, accessibility, availability, and flexible resource management. The specific database service and configuration depend on the requirements of the application.

1. E-Commerce

Online shopping applications can use cloud databases to store and manage:

- Customer Information
- Product Details
- Orders
- Payment-Related Records
- Inventory Information
- Delivery Information

Cloud databases can help e-commerce applications handle changing workloads, such as increased traffic during festival sales or promotional events.

2. Healthcare

Healthcare applications can use cloud databases to manage:

- Patient Records
- Appointment Information

SGOU

- Medical Reports
- Hospital Information
- Laboratory Results

Healthcare databases require strong security, authentication, access control, and data protection because they may contain sensitive information.

3. Education

Educational institutions can use cloud databases for:

- Learner Records
- Course Information
- Examination Results
- Attendance Records
- Learning Management Systems
- Online Examination Systems

For example, a university can use a cloud database to store learner information and provide access to academic applications from different locations.

4. Banking and Finance

Banking and financial applications may use cloud databases to manage:

- Customer Accounts
- Transaction Records
- Loan Information
- Payment-Related Data
- Financial Reports

These applications require high reliability, strong security, appropriate access control, and suitable data consistency mechanisms.

5. Mobile Applications

Mobile applications can use cloud databases to store and manage:

- User Profiles
- Application Data
- User Preferences
- Activity Records

SGOU

- Transaction Information

Cloud databases allow mobile applications to access data through network connections from different locations.

6. Internet of Things (IoT)

IoT systems can generate large volumes of data from connected devices and sensors. Cloud databases can be used to store and manage:

- Sensor Readings
- Device Information
- Location Information
- System Status
- Event Records

For example, a smart monitoring system can continuously send sensor readings to a cloud database for storage and further processing.

7. Business Applications

Organisations can use cloud databases for:

Employee Information

Sales Records

Customer Management

Inventory Management

Business Reports

Cloud databases can support business applications that require access to data from multiple locations.

8. Data Analytics

Cloud databases can be used to store large datasets generated by business applications, websites, IoT devices, and other sources. The stored data can be used for:

- Data Analysis
- Reporting
- Business Intelligence
- Performance Monitoring

SGOU

- Decision Support

Although cloud databases provide benefits such as scalability, accessibility, and flexible resource management, they also have certain challenges. The major challenges are explained below.

1. Security and Privacy

Data stored in the cloud must be protected from unauthorized access, data loss, and security threats.

Organisations need to use appropriate security measures such as:

- Authentication
- Access control
- Encryption
- Network security
- Monitoring and logging

Security responsibilities are generally shared between the cloud provider and the organisation, depending on the selected service.

2. Internet and Network Dependency

Cloud databases generally require network connectivity between the application and the database service.

Poor network connectivity or network failures can cause:

- Increased response time
- Reduced application performance
- Temporary loss of access to the database

Therefore, reliable network connectivity is important for cloud-based applications.

3. Cost Management

Cloud services are often usage-based. Poorly configured resources or unnecessary database usage can increase costs.

Therefore, organisations should monitor resource usage.

Cloud services are often usage-based. Poorly configured resources or unnecessary database usage can increase costs.

SGOU

Therefore, organisations should monitor resource usage.

4. **Vendor Lock-In**

Vendor lock-in occurs when an organisation becomes highly dependent on a particular cloud provider.

Moving applications and databases to another provider may require significant time and effort.

5. **Performance**

Database performance can be affected by:

- Network latency
- Workload
- Database configuration
- Storage performance
- Resource allocation
- Proper database design and monitoring are therefore important.

6. **Data Migration**

Moving an existing database from an on-premises environment to the cloud can be complex.

4.3.8 Popular Cloud Database Providers

Many cloud computing providers offer database services.

Some widely used providers include:

- Amazon Web Services (AWS)
- IBM Cloud
- Microsoft Azure
- Google Cloud

This unit focuses on AWS and IBM Cloud.

Many cloud computing providers offer database services.

Some widely used providers include:

- **Amazon Web Services (AWS)**
- **IBM Cloud**

SGOU

- **Microsoft Azure**
- **Google Cloud**

This unit focuses on AWS and IBM Cloud.

4.3.8.1 Amazon Web Services

Amazon Web Services (AWS) is a major cloud computing platform that provides a wide range of cloud services, including database services.

AWS provides different database options for different application requirements.

Examples include:

- **Amazon RDS**
- **Amazon Aurora**
- **Amazon DynamoDB**
- **Amazon Redshift**

1. **Amazon RDS**

Amazon Relational Database Service (Amazon RDS) is a managed relational database service.

It supports several relational database engines and provides features for database administration, backups, monitoring, and scalability.

It can be used for applications such as:

- Business Applications
- Web Applications
- E-Commerce Systems
- Enterprise Applications

2. **Amazon Aurora**

Amazon Aurora is a managed relational database service designed for cloud applications.

It is compatible with MySQL and PostgreSQL and is designed to provide high performance and availability.

It can be used for applications that require scalable relational database capabilities.

3. **Amazon DynamoDB**

Amazon DynamoDB is a managed NoSQL database service.

It is designed for applications that require low-latency data access and scalable

SGOU

performance.

Typical applications include:

- Web Applications
- Mobile Applications
- Gaming Systems
- IoT Applications

4.3.8.2 IBM Cloud Databases

IBM Cloud provides cloud-based database services for different application requirements.

IBM Cloud database offerings include managed database services that support relational and NoSQL database workloads.

Examples include database services based on technologies such as:

- PostgreSQL
- MySQL
- MongoDB

The exact services and supported features may vary according to the IBM Cloud service selected.

1. **IBM Cloud Databases for PostgreSQL**

PostgreSQL is a popular open-source relational database management system.

IBM Cloud provides managed PostgreSQL services that can be used by applications requiring relational database capabilities.

Applications may include:

Enterprise Applications

Web Applications

Data Management Systems

Business Applications

2. **IBM Cloud Databases for MySQL**

MySQL is a widely used relational database management system.

A managed MySQL service allows organisations to use MySQL in a cloud environment

SGOU

while reducing the amount of infrastructure administration required.

3. **IBM Cloud Databases for MongoDB**

MongoDB is a document-oriented NoSQL database.

It stores data in flexible, document-oriented structures and is suitable for applications that work with flexible or semi-structured data.

4.3.9 AWS and IBM Cloud Database Services

Table 4.4.2: Selected Cloud Database Services

Provider	Database Service / Technology	Database Type	Example Use
AWS	Amazon RDS	Relational	Business and web applications
AWS	Amazon Aurora	Relational	High-performance cloud applications
AWS	Amazon DynamoDB	NoSQL	Mobile and web applications
IBM Cloud	IBM Cloud Databases for PostgreSQL	Relational	Enterprise and web applications
IBM Cloud	IBM Cloud Databases for MySQL	Relational	Web and business applications
IBM Cloud	IBM Cloud Databases for MongoDB	NoSQL	Document-oriented applications

4.3.10 Advantages of Cloud Databases

The major advantages of cloud databases include:

- Reduced Infrastructure Requirements
- Flexible Resource Allocation
- Easy Scalability
- Remote Accessibility
- Backup and Recovery Support
- High Availability Options
- Faster Deployment
- Reduced Infrastructure Maintenance
- Support for Different Database Models
- Integration with Other Cloud Services

SGOU

4.3.11 Limitations of Cloud Databases

Cloud databases also have some limitations:

- Dependence on network connectivity
- Security and privacy concerns
- Possible vendor lock-in
- Usage-based costs
- Migration complexity
- Performance variation
- Requirement for cloud administration knowledge

Therefore, cloud database adoption should be based on the specific requirements of the organisation.



Summarised Overview

SGOU

Cloud databases provide database storage and management capabilities through cloud computing infrastructure. They allow organisations to access database resources without necessarily maintaining physical database infrastructure at their own premises.

Cloud databases can be offered as Database as a Service (DBaaS). In a managed cloud database, the cloud provider handles many infrastructure and administration tasks. In a self-managed cloud database, the organisation manages the database software and many operational activities.

Cloud databases can be relational or non-relational. Relational databases organise data into tables and are suitable for structured information and transactional applications. NoSQL databases include document, key-value, wide-column, and other models designed for specific application requirements.

Cloud databases are widely used in web applications, mobile applications, e-commerce, healthcare, finance, IoT, analytics, and SaaS applications. Important challenges include security, privacy, network dependency, cost management, vendor lock-in, migration, performance, availability, and compliance.

AWS provides database services such as Amazon RDS, Amazon Aurora, and Amazon DynamoDB. IBM Cloud provides database services such as IBM Cloud Databases for MySQL and PostgreSQL, Db2, Cloudant, MongoDB, Redis, and Elasticsearch.

The selection of a cloud database should be based on data structure, workload, scalability, performance, availability, security, cost, and management requirements.

Assignments

1. Explain the fundamentals, characteristics, and need for cloud databases with suitable examples.
2. Explain managed and self-managed cloud databases. Compare their advantages, limitations, and responsibilities.
3. Discuss the major types and use cases of cloud databases.
4. Explain the major challenges associated with cloud databases.
5. Describe the major database services provided by AWS and IBM Cloud.
6. Analyse the factors that should be considered when selecting between a managed and a self-managed cloud database for a web application.

Reference

1. Amazon Web Services. (2026). *Amazon RDS and Aurora documentation*. AWS
2. Elmasri, R., & Navathe, S. B. (2021). *Fundamentals of database systems* (7th ed.). Pearson.
3. Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill.
4. . Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage.
5. Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

SGOU

Suggested Reading

1. IBM. (2026). *IBM Cloud databases*. IBM.

2. Bhupathi, S. **(2025).***Building scalable AI-powered applications with cloud databases: Architectures, best practices and performance considerations*. arXiv.

3. Elmasri, R., & Navathe, S. B. (2021). *Fundamentals of database systems* (7th ed.). Pearson.

4. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill.

5. Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage.

SGOU

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

4
UNIT

Multi-Model Databases

Learning Outcomes

After completing this unit, the learner will be able to:

- explain the concept and features of multi-model databases
- describe the major types of data models supported by multi-model databases
- compare ArangoDB and OrientDB based on their important features and applications
- explain the importance of AI/ML integration in database systems
- analyse the applications of AI/ML techniques in query optimization, predictive analytics, anomaly detection, and smart indexing

SGOU

Background

Modern applications generate and manage different types of data. An e-commerce application, for example, may store customer details as documents, product relationships as graphs, and frequently accessed information as key-value data. Managing these different data models using separate database systems can increase application complexity.

Multi-model databases provide a solution by supporting multiple data models within a single database system. This allows an application to store and process different types of data using one database platform.

At the same time, database systems are becoming more intelligent through the integration of Artificial Intelligence (AI) and Machine Learning (ML). AI/

ML techniques can help databases optimize query processing, identify unusual activities, predict workloads, and select or recommend suitable indexes.

Therefore, understanding multi-model databases and AI/ML integration is important for developing modern data-intensive applications.

Keywords

Multi-Model Database, Data Model, Document Database, Key-Value Database, Graph Database, ArangoDB, OrientDB, Artificial Intelligence, Machine Learning, Query Optimization, Predictive Analytics, Anomaly Detection, Smart Indexing, Supervised Learning, Unsupervised Learning, Reinforcement Learning, Natural Language Processing

Discussion

4.4.1 Introduction

Traditional database systems are generally designed around a particular data model. For example, relational databases organise data into tables, rows, and columns. However, modern applications often deal with different forms of information, such as documents, relationships, key-value pairs, and connected data.

A multi-model database supports multiple data models within a single database system. This allows an application to represent different types of data without necessarily maintaining separate database systems.

For example, consider an online shopping application. Customer information may be stored as documents, product identifiers may be accessed using key-value operations, and relationships between customers and products may be represented using graphs. A multi-model database can support these different ways of representing and accessing data.

In addition to flexible data storage, modern databases increasingly use AI and ML techniques to improve database operations. Machine learning can be applied to areas such as query optimization, indexing, workload prediction, and anomaly detection. Recent research also examines the integration of ML into query optimizers, indexing structures, self-tuning mechanisms, and anomaly detection systems.

SGOU

4.4.2 Multi-Model Databases

Multi-model databases are database systems that support multiple data models within a single database system. They provide a common database environment for managing different types of data according to application requirements.

A multi-model database may support data models such as document, key-value, graph, relational, or other models, depending on the database product. This flexibility allows different types of structured and semi-structured data to be managed within a single database system. It can also reduce the need to use separate database systems for different types of data.

Multi-model databases commonly provide functions such as data storage, querying, indexing, backup, and recovery. However, the available features and supported data models vary among different database products. An organisation can select a suitable multi-model database based on its application requirements and data management needs.

Multi-model databases can:

1. Support multiple data models.
2. Reduce the need for multiple database systems.
3. Simplify application development.
4. Provide flexibility in data representation.
5. Support complex relationships between data items when a graph model is available.
6. Handle different application requirements.
7. Reduce the complexity of integrating separate databases.

Example

Consider a hospital management system.

It may contain:

- Patient Records
- Doctor Information
- Medical Reports
- Relationships Between Patients and Doctors
- Treatment History

SGOU

Different data models may be useful for different parts of the system. For example, patient records and medical reports can be represented using a document model, while relationships between patients and doctors can be represented using a graph model. A multi-model database can provide a common database environment for managing these different types of data.

Steps for Implementing a Multi-Model Database

Implementing a multi-model database requires proper planning.

1. **Identify the Data Requirements:** Identify the types of data that need to be stored and managed.

2. **Select Appropriate Data Models:** Decide which models, such as document, graph, key-value, or relational, are required.

3. **Select a Suitable Database:** Choose a multi-model database based on the application's requirements and supported data models.

4. **Plan the Database Architecture:** Design the database to support efficient data storage, retrieval, and analysis.

4.4.2.1 Features of Multi-Model Databases

The important features of multi-model databases are explained below.

1. **Data Storage, Backup, and Recovery**

The main purpose of a multi-model database is to store and manage data. It can store data in different forms, such as documents, graphs, key-value pairs, and other supported formats. Depending on the database system, facilities for creating backups and recovering data may also be provided.

2. **Querying and Indexing**

Multi-model databases provide querying mechanisms for retrieving and managing data. They also use indexes to improve the speed of data retrieval. Depending on the database system, different data models may have different query and indexing methods.

3. **ACID Transactions**

Many multi-model databases support ACID transactions. ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties help maintain the correctness and reliability of data during database operations. They are especially important when errors, system failures, or unexpected interruptions occur.

4. **Integration**

Multi-model databases allow different data models to be managed within a single database system. This reduces the need to maintain separate databases for different

SGOU

types of data. They may also support the integration of data from different sources and formats, depending on the capabilities of the database system.

5. **Security**

Security is an important requirement in database systems. Multi-model databases may provide features such as authentication, authorization, encryption, auditing, and access control. These features help protect stored data from unauthorized access and support secure database management.

6. **Flexible Data Management**

Multi-model databases provide flexibility in choosing the appropriate data model for a particular application. For example, a document model can be used for document-based information, while a graph model can be used to represent relationships between entities. This flexibility makes multi-model databases suitable for applications with diverse data requirements.

4.4.3 Types of Data Models in Multi-Model Databases

A multi-model database can support several data models.

1. **Document Model**

Data is stored as documents, commonly using JSON-like structures.

Example

```
{
 "Learner_ID": 101,
 "Name": "Anu",
 "Course": "MCA"
}
```

The document model is useful when data has a flexible or changing structure.

Applications

- Content Management
- Product Catalogues
- User Profiles
- Web Applications

SGOU

2. Graph Model

The graph model represents data using:

- Nodes
- Edges
- Properties

For example:

Learner → Enrolled in → Course

Here:

- Learner is a node.
- Course is a node.
- Enrolled in is an edge or relationship.

Graph models are useful when relationships between data items are important.

Applications

- Social Networks
- Recommendation Systems
- Fraud Detection
- Network Analysis

3. Key-Value Model

Data is stored as a pair consisting of a key and its corresponding value.

101 → Anu

102 → Rahul

103 → Meera

Key-value models are useful when data needs to be accessed quickly using a unique key.

Applications

- Caching
- Session Management
- User Preferences

SGOU

4. **Relational Model**

The relational model represents information using tables containing rows and columns.

Learner_ID	Name	Course
101	Anu	MCA
102	Rahul	MCA

The relational model is widely used for structured data and transaction-oriented applications.

4.4.4 Advantages of Multi-Model Databases

Multi-model databases provide several advantages.

1. **Flexibility:** Different data models can be used according to the requirements of the application.
2. **Reduced System Complexity:** A single database platform can manage different types of data.
3. **Faster Application Development:** Developers may not need to integrate several independent database systems.
4. **Support for Complex Applications:** Applications containing different types of data can be developed using a common database environment.
5. **Better Data Integration:** Different representations of related data can be managed within the same database environment.
6. **Cost and Maintenance Benefits:** Using fewer independent database systems may reduce administration and maintenance requirements.

4.4.5 Limitations of Multi-Model Databases

Although multi-model databases provide flexibility, they also have some challenges.

1. **Increased System Complexity:** Supporting multiple data models can make database design and administration more complex.
2. **Learning Requirements:** Database administrators and developers may need to understand different data models and query approaches.
3. **Performance Differences:** A database may not provide the same level of performance for every data model or workload.

SGOU

4. **Vendor Dependency:** Applications may become dependent on the features and query mechanisms provided by a particular database product.

5. **Migration Challenges:** Moving applications from one multi-model database system to another may require changes to data structures and application logic.

4.4.6 Popular Multi-Model Databases

Several database systems support multiple data models. Two important examples are:

1. **ArangoDB**

2. **OrientDB**

4.4.6.1 ArangoDB

ArangoDB is a native multi-model database that supports document, graph, and key-value data models. It uses JSON documents as an important part of its data representation. ArangoDB uses AQL (ArangoDB Query Language) for querying data.

Example

Consider a learning management system.

A learner can be represented as a document:

```
{
 "id": 101,
 "name": "Anu",
 "course": "MCA"
}
```

The relationship between the learner and the course can also be represented using graph concepts.

Features of ArangoDB

- Document Database Capabilities
- Graph Database Capabilities
- Key-Value Operations
- JSON-Based Data Representation

SGOU

- Graph Traversal
- Querying and Filtering
- Indexing
- Support for Distributed Applications

ArangoDB allows graph vertices and edges to be represented as documents. Edge documents contain references that identify their source and target documents.

Applications

ArangoDB can be used in:

- Recommendation Systems
- Knowledge Graphs
- Fraud Detection
- Content Management
- Network Analysis
- Web Applications

4.4.6.2 OrientDB

OrientDB is a multi-model database that combines document and graph database capabilities.

It supports the representation of information as documents and allows relationships between entities to be represented using graph concepts.

Example

Consider a social networking application.

A user can be represented as a document:

```
{
 "Name": "Anu",
 "Age": 23
}
```

Relationships can represent:

Anu → follows → Rahul

SGOU

This combination allows the database to manage both entity information and relationships.

Features of OrientDB

- Multi-Model Database Support
- Document Model
- Graph Model
- Support for Relationships
- Flexible Schema
- Suitable for Interconnected Data

Applications

OrientDB can be used in:

- Social Networking
- Recommendation Systems
- Knowledge Management
- Network Analysis
- Content Management

4.4.7 ArangoDB and OrientDB Comparison

Table 4.4.1: Comparison of ArangoDB and OrientDB

Feature	ArangoDB	OrientDB
Database type	Multi-model database	Multi-model database
Document support	Yes	Yes
Graph support	Yes	Yes
Key-value support	Yes	Supports key-value style operations through its multi-model capabilities
Query approach	AQL(ArangoDB Query Language)	SQL-like query language and graph operations
Main strength	Combining documents and graphs	Combining document and graph models

SGOU

Typical applications	Recommendations, knowledge graphs, fraud detection	Social networks, knowledge management, interconnected data

4.4.8 AI and ML Integration in Databases

AI/ML integration in databases refers to the use of Artificial Intelligence (AI) and Machine Learning (ML) techniques to improve database operations, analyse stored data, automate database management tasks, and generate useful predictions from data.

Traditionally, database systems mainly focused on storing, retrieving, and managing data. Modern database systems can also use machine learning techniques to make certain database operations more adaptive and efficient.

For example, a database system may learn from previous query workloads and use the learned information to improve query execution or indexing decisions.

AI/ML techniques can be applied to areas such as query optimization, indexing, self-tuning, anomaly detection, workload forecasting, and performance diagnostics.

4.4.9 Importance of AI/ML Integration in Databases

AI and ML integration can make modern database systems more adaptive and efficient. AI/ML techniques can assist with tasks such as workload analysis, query optimization, indexing, security monitoring, and resource management. Some database systems can also automate selected maintenance and administrative tasks based on their capabilities and configuration.

AI/ML can provide several benefits to database systems.

1. **Improved Performance**

Machine learning algorithms can analyse database workload patterns and help optimize system performance and resource utilization.

2. **Query Optimization**

ML techniques can improve query execution plans by learning from previous query execution patterns instead of relying only on predefined optimization rules.

3. **Predictive Analytics**

Historical access and usage patterns can be used to build predictive models that help forecast data trends, traffic spikes, and resource requirements.

4. **Anomaly Detection**

ML models can identify unusual access patterns or performance deviations, which can help detect potential security threats, suspicious activities, or system problems.

SGOU

5. **Intelligent Indexing**

Machine learning can analyse incoming queries and workload patterns to recommend suitable indexes or identify indexes that may no longer be required. Learned index structures can also be used in some database systems to improve data access efficiency.

6. **Automation**

AI/ML techniques can assist in automating selected database administration tasks, such as capacity planning, resource management, workload tuning, and performance monitoring, depending on the capabilities of the database system.

4.4.10 Key Areas of AI/ML Integration in Databases

4.4.10.1 Query Optimisation

Query optimization is the process of finding an efficient way to execute a database query.

A query may be executed using different strategies. The database system needs to select a suitable execution plan.

Machine learning can learn from previous queries and workload patterns to help predict which execution plan may perform better.

Example

Suppose an e-commerce database receives thousands of product-search queries every day. A machine learning model can learn from historical query performance and help identify efficient execution strategies for frequently occurring queries.

4.4.10.2 Predictive Analytics

Predictive analytics uses historical data, statistical algorithms, and machine learning techniques to estimate future outcomes. Modern database systems can provide the storage and processing capabilities required for the large volumes of historical data used by predictive models.

An online shopping system can analyse historical transaction data to predict:

- Future Product Demand and Inventory Requirements
- Seasonal Sales Spikes and Purchasing Trends
- Customer Churn Probability and Buying Behaviour

By integrating analytics with data storage and processing, the database can become an active source for predictive insights rather than simply storing historical information.

SGOU

4.4.10.3 Anomaly Detection

Anomaly detection is the process of identifying data points, access patterns, or query behaviours that deviate significantly from established normal behaviour.

An anomaly within a database environment may indicate risks such as:

- Financial Fraud or Fraudulent Transactions
- Security Threats and Unauthorized Access
- Hardware or Database System Failures
- Data Quality Problems

Example

Suppose a banking database normally records small, localized daily transactions for a particular user account. If a very large financial transfer suddenly originates from an unusual international IP address, an integrated anomaly detection model can flag the transaction for further investigation and may alert the security system.

Applications

- Banking Fraud Prevention
- Real-Time Cybersecurity and Intrusion Detection
- Network Traffic and Server Log Monitoring
- Financial Risk Management and Compliance Auditing
- System Fault and Deadlock Detection

4.4.10.4 Smart Indexing

An index is a specialized data structure that helps accelerate data retrieval without scanning every row in a table. Traditionally, determining which columns require indexing relies largely on database administrators (DBAs) and workload analysis.

Smart indexing uses workload analysis and machine learning to evaluate query patterns and can recommend suitable indexes or identify indexes that may no longer be required. Some advanced implementations also use learned index structures, which use statistical or machine-learning models to improve data access for particular workloads.

Example

If a university database frequently executes queries containing the condition:

WHERE Course = 'MCA', an intelligent indexing system can detect the recurring query pattern, evaluate the potential benefit, and recommend or create an index on the Course column, depending on the capabilities of the database system.

SGOU

4.4.11 Machine Learning Techniques Used with Databases

Different ML techniques can be applied to different database-related problems.

The major techniques included in this unit are:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning
4. Natural Language Processing

4.4.11.1 Supervised Learning

Supervised learning is an ML approach in which a model is trained using labelled datasets containing known input-output mappings.

$$\text{Input } (x) \longrightarrow \text{Model } f(x) \longrightarrow \text{Known Output } (y)$$

The algorithm learns the relationship between the input data and the known output labels. The trained model can then be used to make predictions for new or unseen data.

Example: A database contains historical credit card transactions labelled as Legitimate or Fraudulent. A supervised learning model can be trained using these historical transactions and then used to classify new transactions as legitimate or potentially fraudulent.

Applications in Databases

- Transactional Fraud Classification
- Predictive Resource Utilization
- Query Execution Time Prediction
- Customer Churn Classification

4.4.11.2 Unsupervised Learning

Unsupervised learning operates on unlabelled datasets. Instead of predicting predefined target values, the model analyses the data to discover hidden patterns, clusters, or underlying structures.

Example: An e-commerce database stores customer purchase histories without predefined category labels. An unsupervised clustering algorithm, such as k-Means,

SGOU

can group customers into distinct market segments based on similarities in their purchasing behaviour. These segments can then be used for targeted marketing campaigns.

Applications in Databases

- Automated Customer Segmentation
- Unsupervised Anomaly Detection (Outlier Discovery)
- Database Access Pattern Exploration
- Data Quality and Schema Analysis

4.4.11.3 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning technique in which an agent learns by interacting with an environment. The agent takes an action, observes the resulting state, and receives a positive reward or negative penalty based on the outcome.

Example in Database Systems

An RL system can experiment with different query execution strategies. If a strategy results in faster query execution, the system can receive a positive reward. Over time, the system can learn which strategies are more effective for different workloads.

An RL agent is used for database query optimization. It experiments with different execution plans for incoming queries. If a selected plan reduces execution time, the agent receives a positive reward signal. Through repeated interactions, the agent learns effective execution strategies for different workloads.

Applications in Databases

- Adaptive Query Execution and Join-Order Optimization
- Dynamic Database Memory and Buffer Pool Allocation
- Automatic Parameter Tuning and System Configuration
- Intelligent Workload Management and Scheduling

4.4.11.4 Natural Language Processing

Natural Language Processing (NLP) enables database systems to understand human language queries and translate them into formal database query languages, such as SQL.

Example

A non-technical administrator enters a plain English query:

"Find all MCA learners above 23 years of age."

SGOU

The NLP-based system interprets the query as:

- **Target Table:** Learner
- **Filter Condition 1:** Course = 'MCA'
- **Filter Condition 2:** Age > 23

It can then generate the corresponding SQL query:

SELECT * FROM Learner WHERE Course = 'MCA' AND Age > 23;

Applications in Databases

- Conversational Natural Language Query Interfaces (Text-to-SQL)
- Intelligent database chatbots and virtual assistants
- Voice-driven data interaction modules
- Automated enterprise Q&A reporting engines

4.4.12 Applications of AI/ML in Databases

AI and ML integration has applications in many areas.

1. **Healthcare**

Healthcare databases contain patient records, medical reports, laboratory results, and other information.

AI/ML can be used for:

- Predictive Healthcare Analytics
- Patient Risk Prediction
- Anomaly Detection
- Medical Data Analysis

2. **Finance**

Financial databases contain transaction records, customer information, and market information.

AI/ML can be used for:

- Fraud Detection
- Credit Risk Prediction
- Transaction Analysis

SGOU

- Anomaly Detection
- Financial Forecasting

3. E-Commerce

E-commerce databases store information about customers, products, orders, and transactions.

AI/ML can be used for:

- Product Recommendation
- Customer Segmentation
- Demand Prediction
- Fraud Detection
- Personalised Services

4. Cybersecurity

AI/ML can analyse database and network activity to identify unusual behaviour.

Applications include:

- Intrusion Detection
- Suspicious Activity Detection
- Access Pattern Analysis
- Security Monitoring

5. Database Administration

AI can assist database administrators in:

1. Monitoring Performance
2. Identifying Bottlenecks
3. Recommending Indexes
4. Predicting Resource Requirements
5. Optimising Workloads

SGOU

4.4.13 Relationship Between AI/ML Techniques and Database Applications

Table 4.4.2: AI/ML Techniques and Applications

Technique	Major Database Application
Supervised Learning	Fraud detection, prediction, classification
Unsupervised Learning	Customer segmentation, pattern discovery
Reinforcement Learning	Query optimisation, resource management
NLP	Natural language queries and database assistants
Machine Learning	Smart indexing and performance prediction

Summarised Overview

A multi-model database supports different data models within a single database system. It is useful for handling different types of data according to application requirements. Common data models include document, key-value, graph, relational, and object-oriented models. In a document model, data is stored as flexible documents. In a key-value model, data is stored as key-value pairs. In a graph model, data is represented using nodes, relationships, and properties.

ArangoDB supports document, graph, and key-value data models. OrientDB supports document and graph models and provides additional multi-model capabilities. AI and Machine Learning can be integrated with databases to make database operations more intelligent. They can help with query optimization, predictive analytics, anomaly detection, and smart indexing. Supervised Learning, Unsupervised Learning, and Reinforcement Learning can be used for different database-related tasks. Natural Language Processing (NLP) can also help users interact with databases using natural language.

These technologies are useful in areas such as healthcare, finance, e-commerce, cybersecurity, and recommendation systems. However, issues such as data quality, security, privacy, accuracy, computational cost, and explainability must be considered. Overall, multi-model databases provide flexible data management, while AI and ML provide intelligent analysis and automation.

SGOU

Assignments

1. Explain the concept, features, and need for multi-model databases with suitable examples.
2. Explain the document, key-value, and graph data models with suitable examples.
3. Describe the important features and applications of ArangoDB and OrientDB.
4. Explain the importance of AI/ML integration in modern database systems.
5. Explain query optimization, predictive analytics, anomaly detection, and smart indexing in AI-enabled databases.
6. Compare multi-model databases and traditional single-model databases and discuss their suitability for modern applications.

Reference

1. .https://decidesoftware.com/top-multi-model-databases/
2. Panda, S. P. (2025).Relational, NoSQL, and artificial intelligence-integrated database architectures: Foundations, cloud platforms, and regulatory-compliant systems. Deep Science Publishing.
3. Xiong, X., Yu, J., & He, Z. (2025). AutoQuo: An adaptive plan optimizer with reinforcement learning for query plan selection. Knowledge-Based Systems.
4. Elmasri, R., & Navathe, S. B. (2017). Fundamentals of database systems (7th ed.). Pearson.
5. Date, C. J., Kannan, A., &Swamynathan, S. (2009). An introduction to database systems (8th ed.). Pearson Education.

SGOU

Suggested Reading

1. Official documentation for scikit-learn model persistence, Flask, and the machine learning services of AWS, Azure, and Google Cloud.

2. Official scikit-learn documentation on preprocessing and feature scaling, for a complete reference on MinMaxScaler, StandardScaler, RobustScaler, and encoding tools.

3. Osborne, J. W. (2012). Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data. Sage Publications.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SGOU

MODEL QUESTION PAPER SETS

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……

SET-1

Reg. No : …….....
Name : ……...

MASTER OF COMPUTER APPLICATION, SECOND SEMESTER
END SEMESTER EXAMINATION

DISCIPLINE CORE COURSE
M25CA05DC DATABASE TECHNOLOGIES (CBCS - PG)
2025-26 - Admission

Time: 3 Hours Max Marks:70

SECTION A

Answer **any 10** questions each carries two marks. 10x2=20

1. Differentiate between structured data and semi-structured data.
2. What is data independence in a DBMS?
3. Define a candidate key.
4. What is a partial key in a weak entity?
5. Define First Normal Form (1NF)
6. What is a foreign key? State its purpose
7. What is the use of the ORDER BY clause?
8. What is the purpose of the IN operator?
9. What is variable in PL/SQL?
10. Explain the Implicit Cursors.
11. What is REF Cursors
12. What is the Trigger?
13. Mention any four characteristics of NoSQL databases.
14. What is a vector database? Mention one of its applications.
15. What is a multi-model database? Mention any two data models it supports.

SGOU

SECTION-B

Answer **any 6** questions each carries 5 marks**.** 6x5=30

16. Explain the different levels of database abstraction with suitable examples.

17. Explain how a many-to-many relationship in an ER model is mapped to relational tables.

18. Explain partial dependency and transitive dependency with suitable examples.

19. Explain the CREATE, ALTER, DROP, and TRUNCATE commands with examples.

20. Explain the different sections of a PL/SQL block.

21. Explain Statement-Level and Row-Level Triggers

22. Explain the popular cloud database providers.

23. Explain the structure of a graph database with suitable examples of nodes, relationships, and properties.

24. Explain the different types of data models supported by multi-model databases with suitable examples.

SECTION C

Answer **any 2** questions each carries ten marks 2x10=20

25. Explain the process of converting an ER model into a relational database schema, covering the mapping of entities, relationships, weak entities, and generalization and specialization.

26. Explain SQL aggregate functions in detail. Discuss COUNT(), SUM(), AVG(), MIN(), and MAX() with suitable examples, including the handling of NULL and DISTINCT.

27. What are control structures? Explain the different types of control structures in PL/SQL.

28. Explain the fundamentals of cloud databases and discuss the different types of cloud databases

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……

SET-2

Reg. No: ……..
Name : ……..

MASTER OF COMPUTER APPLICATION, SECOND SEMESTER
END SEMESTER EXAMINATION

DISCIPLINE CORE COURSE
M25CA05DC DATABASE TECHNOLOGIES (CBCS - PG)
2025-26 - Admission

Time: 3 Hours Max Marks: 70

SECTION A

Answer **any 10** questions each carries two marks 10x2=20

1. What is a database schema?
2. List any four characteristics of a DBMS.
3. Define a composite key with an example.
4. What is the purpose of Data Definition Language (DDL)?
5. What are the uses of the MIN() and MAX() functions?
6. What iWs the purpose of the SUBSTRING() function?
7. Write the general syntax for creating a view in SQL
8. State the conditions required for a relation to be in Second Normal Form (2NF)
9. Explain the different data types in PL/SQL.
10. Explain explicit cursors in PL/SQL.
11. What is Stored Procedure?
12. Write the syntax for creating a trigger in PL/SQL.
13. Mention any two cloud database services offered by AWS and any one offered by IBM Cloud.
14. Mention any four characteristics of document stores.
15. What is a column-family store?

SGOU

SECTION-B

Answer **any 6** questions each carries 5 marks . 6x5=30

16. Explain the different types of DBMS based on their data models.

17. Explain the different types of database users and their roles.

18. Explain database normalization and discuss the insertion, deletion, and update anomalies with examples

19. Explain the use of comparison operators, logical operators, and the IN operator in SQL queries

20. Differentiate between Stored Procedures and Functions in PL/SQL

21. Write about the components of a trigger in PL/SQL.

22. Explain the four major types of NoSQL databases.

23. Explain the features and applications of ArangoDB and OrientDB.

24. Explain the challenges of vector databases and the role of embeddings in vector databases.

SECTION C

Answer **any 2** questions each carries ten marks 2x10=20

25. Explain the Entity-Relationship (ER) model with a suitable example. Discuss entities, attributes, relationships, key constraints, and the different types of keys used in the ER model.

26. Explain SQL string functions in detail with suitable examples for each function.

27. What is trigger?Explain different types of triggers.

28. Explain vector indexing and discuss the different vector indexing techniques.

SGOU

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ്വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kollam
Sreenarayanaguru Open University
Headquarters Building
BSNL Bhavan, Vellayittambalam,
Kollam, Kerala, Pin : 691012
Ph: 9188912554
email:rckollam@sgou.ac.in

Tripunithura
Govt. College
Tirpunithura, Ernakulam
Kerala, Pin : 682301
Ph: 04842927436
email:rcedirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery
Kannur, Pin : 670106
Ph: 04952990494
email:rctdirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad
Kerala, Pin : 679303
Ph: 04662912009
email:rcpdirector@sgou.ac.in

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin : 673002
Ph: 04952920228
email:rckdirector@sgou.ac.in

SAY NO TO RAGGING

BEFORE YOU EVEN THINK OF RAGGING



Humiliation

Blacklisting

Ruined Career

Download

ANTI RAGGING

App

Don't just stand and watch. Stop Ragging! Show Character

Visit UGC Website i.e. www.ugc.Ac.in & www antinaging.in to UGC Anti Ragging regulations.

Ministry of Education
Government of India

Are You Being Ragged ?

विश्वविद्यालय अनुदान आयोग
University Grants Commission
quality higher education for all

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY



SREENARAYANAGURU OPEN UNIVERSITY

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

SGOU

Database Technologies

COURSE CODE: M25CA05DC

SGOU

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-181574-1-0