Mẫu 6.2/Template 6.2

TRƯỜNG ĐẠI HỌC KHOA HỌC VÀ CÔNG NGHỆ HÀ NỘI

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

Fundamentals of Databases

Editors: NGUYEN Hoang Ha

LE Huu Ton

TRƯỜNG ĐẠI HỌC KHOA HỌC VÀ CÔNG NGHỆ HÀ NỘI

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

Fundamentals of Databases

Editors: NGUYEN Hoang Ha

LE Huu Ton

HANOI - 2024

Preface

Welcome to "Fundamentals of Databases", a comprehensive guide designed to introduce the world of databases. Whether you are either undergraduate students in computer science, information technology, data science, cyber security, and related fields or a professional seeking to enhance your knowledge, this book aims to equip you with the essential concepts and practical skills in the field of databases. Prior knowledge of basic computer science concepts is beneficial but not mandatory, as the book starts with fundamental topics and gradually progresses to more advanced subjects.

The book covers topics described in the official syllabus of the subject Fundamentals of Databases at the Bachelor program at USTH. It contains the basic concepts of database world and especially the relational databases. Readers are still encouraged to consult other materials for in depth explanations.

During the journey through this book, to clarify and illustrate the concepts, we use a continuous case study of an Academic Management System, which is very familiar to students and the academic community. The content is organized into five chapters, each focusing on a key aspect of database systems:

- Introduction to Databases: This chapter sets the stage by introducing the fundamental concepts of databases. It covers the importance of databases, the users, the history and the current development trends of database technology.
- The Relational Model: We delve into the relational model, which is the most widely used database model today. Topics include relations, attributes, keys, integrity constraints, and relational algebra. This chapter provides a thorough understanding of how data is structured and manipulated in relational databases.
- Structured Query Language: This chapter covers the basics of SQL, which is the standard language for interacting with relational databases. Topics such as coding with SQL, the taxonomy and syntaxt of SQL statements are also explored to give readers a robust understanding of SQL.
- Non-table Objects in RDBMS: Beyond tables, relational database management systems (RDBMS) include various non-table objects such as indexes, views, stored procedures, and triggers. This chapter discusses the purpose and usage of these objects, illustrating how they enhance database functionality and performance.
- Analyze and Design: This final chapter focuses on the methodologies for analyzing and designing databases. Topics include requirements analysis, data analysis using ER diagrams, logical and physical designs, and normalization

considerations. This chapter provides practical guidelines for creating efficient and scalable database designs**.**

This textbook is a labor of love by Dr. NGUYEN Hoang Ha and Dr. LE Huu Ton. The authors express our gratitude to Assoc. Prof. Lương Chi Mai, Assoc. Prof. Phạm Thanh Giang, and Dr. Hoàng Đỗ Thanh Tùng for their valuable and constructive comments during the improvement jury. We hope that "Fundamentals of Databases" serves as a valuable resource for your academic and professional growth.

Authors

Table of Contents

Glossary

Terminology	Explanation
DBMS - Database Management System	A kind of software to facilitate creation, organization, storage, retrieval, and management of data in databases.
Database system	The combination of data, DBMS, and the associated applications.
Relational Data Model	A method to organize data into tables (relations) of rows and columns.
Attribute (the Relational Data Model)	A column in a table that represents a specific property or characteristic of the data stored in that table.
Tuple (the Relational Data Model)	A row in a table, representing a single record that contains a set of related data values for each attribute (column) in the table.
Domain (the Relational Data Model)	a domain is the set of all possible values that an attribute (column) can take
Functional Dependancy	Relationiship between two sets of attributes where the value of one uniquely determines the value of another.
Entity	An entity is a distinct object or concept.
Relationship	Defines how entities are related to each other in the database.
Attribute	A characteristic or property of an entity.
Cardinality	Cardinality specifies the number of instances of one entity that can or must be associated with each instance of another entity in a relationship.
Schema	overall structure or blueprint of a database, including the logical organization of tables, their attributes (columns), and the relationships between them
Entity-Relationship Model	A conceptual framework used to describe the relationships between entities in a database
SQL (Structured Query Language)	A standardized programming language used to manage and manipulate relational databases.
DDL - Data Definition Language	SQL statements used to define and manage the structure of databases.
DML - Data Manipulation Language	SQL statements used to query, insert, update, and delete data from databases.
DCL - Data Control	SQL statements used to manage access permissions and

Language	privileges in a database.
TCL - Transaction Control Language	SQL keywords used to manage manages transactions in a database.
PSM - Persistent Stored Module	Stored procedures or functions within a DBMS.
Index	Data structure that improves the speed of data retrieval operations.
View	A virtual table derived from one or more base tables.
Stored Procedure	A set of SQL statements stored in the database and executed as a single unit.
Trigger	A kind of special stored procedure that automatically executes in response to specified events.
Relationship	In the context of databases, a relationship defines how two or more entities are related to each other,
SDLC - System Development Life Cycle	A structured approach to software development encompassing planning, design, development, testing, deployment, and maintenance phases.
AMS - Academic Management System	The sample fictitious software system for demonstrating concepts and techniques of Relational databases used in this textbook.
ERD - Entity Relationship Diagram	A visual representation of data used for business analysis.
1NF	First Normal Form
2FN	Second Normal Form
3NF	Third Normal Form

Chapter 1: Introduction to Databases

Databases have become indispensable for businesses across all sectors [1]. Whether people are browsing through well-known websites or numerous smaller information-providing platforms, databases play a crucial role in delivering the requested information. Corporations rely on databases to store their vital records securely. Some typical applications that rely on databases include customer relationship management systems, accounting management systems, and online shopping platforms. Moreover, databases serve as foundational tools in various scientific endeavors, from gathering astronomical data to studying the human genome and exploring protein properties in biochemistry, among numerous other scientific pursuits.

This chapter starts with the introduction about the domain of databases, tracing their evolution from humble beginnings to the cutting-edge technologies of today. From the pioneering efforts of early data processing systems to the emergence of relational databases, NoSQL solutions, and beyond, we witness the relentless march of progress driven by technological advancements and evolving demands.

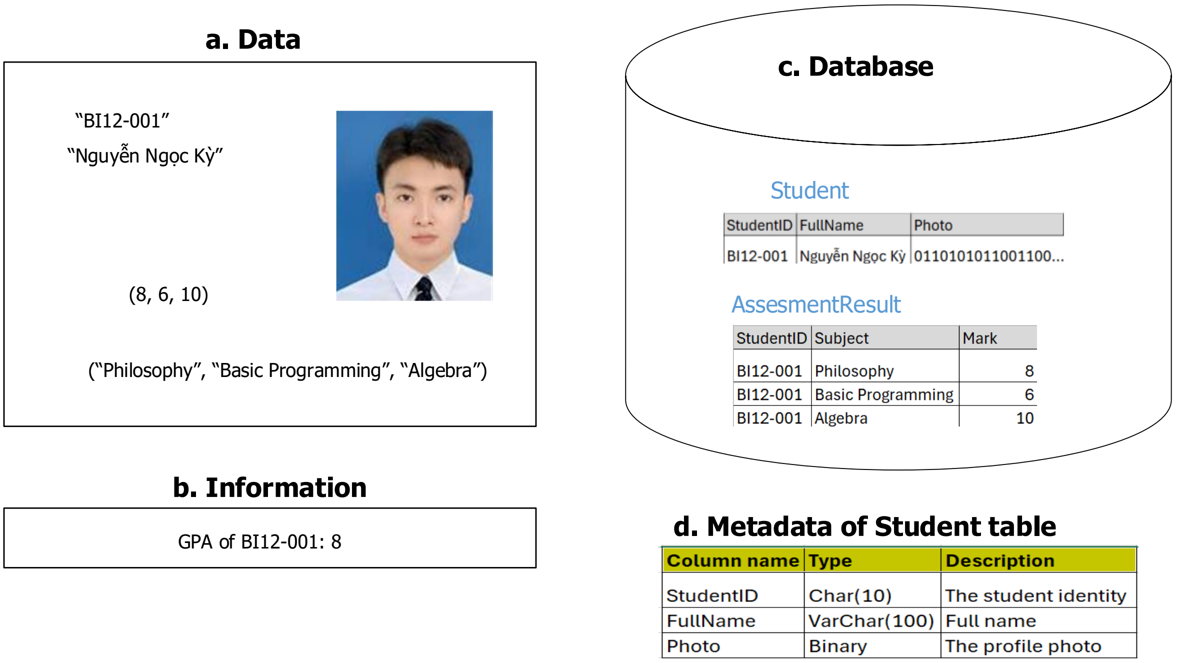
By the end of this chapter, the learners should have a solid understanding of the database world, specially achieve the following goals:

- Understand the key concepts in the database world.
- Be aware of the importance of studying databases.
- Distinguish different types of database user.
- Get milestones in the database history.
- Identify current trends in databases.

1. Basic Concepts and Definitions

Database domain is large with a long history of development. In order to help readers embark on a journey through the world of databases, this section aims to provide a comprehensive understanding of essential concepts in the domain.

The term database has “data” as its prefix, so we first need to clarify this terminology. **Data** *refers to raw facts, figures, and statistics of objects or events*. It can be in the form of numbers, text, images, audio, or any other format. For examples there are data on a student transcript (see Figure 1a.): strings “BI12-001” and “Nguyễn Ngọc Kỳ” represent the student ID and full name of a student respectively, a list of strings describing subject names (“Philosophy”, “Basic Programming”, “Algebra”), a list of numbers (8, 6, 10) shows the test scores of a student on these subjects, the photo of that student.



a. Data

b. Information

c. Database

Student

StudentID	FullName	Photo
BI12-001	Nguyễn Ngọc Kỳ	0110101011001100...

AssesmentResult

StudentID	Subject	Mark
BI12-001	Philosophy	8
BI12-001	Basic Programming	6
BI12-001	Algebra	10

d. Metadata of Student table

Column name	Type	Description
StudentID	Char(10)	The student identity
FullName	VarChar(100)	Full name
Photo	Binary	The profile photo

Figure 1. Examples about data, information, database, and metadata.

Concerning the organization, data can be categorized into three main types: structured, unstructured, and semi-structured. The data structured is organized in a predefined manner, often in tabular or hierarchical formats, making it easy to search and analyze. The spreadsheet is a good example of structure data. In contrast, unstructured data lacks a specific format or structure, making it more challenging to process and analyze. Examples include text documents, images, and videos. The semi-structured data does not conform to a rigid structure but contains tags or markers to separate elements. Examples include XML and JSON files.

Data can be considered as the material from which Information is derived so data by itself needs more context and meaning once it is processed and analyzed. Readers should distinguish between the data and information concepts. More formally, **Information** *is processed, organized, and structured data that has context, relevance, and meaning*. It provides insight or knowledge when interpreted. Information is obtained by analyzing and interpreting data. For example, from the scores of students, we can infer his GPA, which is 8 (Figure 1b.). This information helps us understand the overall academic results of that student.

Prior to the existence of computers, data was managed and processed manually. Businesses and organizations relied on paper records, ledgers, and filing systems to store and manage information. This method was labor-intensive, error-prone, and inefficient, especially as the volume of data grew. Electronic computers have progressively transformed data management from manual processes to sophisticated, automated systems. The Electronic Numerical Integrator and Computer (ENIAC), developed in 1945, was among the first electronic computers designed to handle complex numerical calculations. Data was input via punched cards and outputs were displayed using lights and printouts. The Universal Automatic Computer I (UNIVAC I) was the first commercially produced computer in the United States used magnetic tape for data storage, which allowed for the storage and retrieval of larger datasets compared to punched cards. Throughout the evolution of computers, numerous methods and technologies have been developed for data management and processing, reflecting the fact that working with data is always one of the most important tasks of computers.

Recent years have seen a rapid growth of data, which is often referred to as the "data explosion" or "data deluge". The sheer volume of data generated daily is staggering. According to recent estimates, over 2.5 quintillion bytes of data are created every day, and this number continues to grow. There are some main factors contributing to this exponential growth. First of all, we are in the era of Digital Transformation, which means more and more processes, transactions, and interactions are being digitized, leading to a vast increase in the volume of digital data generated by businesses, organizations, and individuals. Secondly, the proliferation of IoT devices such as sensors, smart devices, and connected machines generates enormous amounts of data from various sources, including environmental sensors, wearable devices, industrial equipment, and more. Third, the widespread use of social media platforms, e-commerce websites, online streaming services, and other digital platforms results in a continuous stream of user-generated content, interactions, and transactions, contributing significantly to the growth of data. The rapidly expanding era of data is known as the "Age of Data" where data has become a valuable asset in nearly every aspect of life. In this era, data is more than just a byproduct of digital interactions; it has become a crucial asset assets and valuable for a variety of applications, from traditional to modern. Data is driving decision-making, innovation, and competitive advantage across various domains. In this context, data science has emerged as an interdisciplinary field that enables the effective utilization of the massive amounts of data generated in this era. It encompasses a range of techniques and methodologies to transform raw data into valuable insights.

These facts suggest that efficiently managing data has always been a crucial aspect of computers, from the past to the present and likely into the future. For that purpose, a ***database** is an organized collection of data that allows for efficient storage, retrieval, management, and manipulation*. According to Oracle, A database is an organized collection of structured information, or data, typically stored electronically in a computer system. Figure 1c. illustrates database in that sense: we put all data on the student transcript into 02 tables. Databases are designed to handle large volumes of data systematically and securely. They play a crucial role in various applications and services, from business operations to scientific research.

Databases have become indispensable for businesses across all sectors. Whether you are browsing through well-known websites like google.com, amazon.com, vnexpress.net, or numerous smaller information-providing platforms, databases play a foundational role in delivering the requested information. Here are some domains of applications illustrating their widespread use:

- Enterprise: databases are integral to the management, organization, and utilization of vast amounts of data generated and processed by businesses and organizations. Corporations rely on databases to store their vital records securely. Common types of enterprise applications include accounting, customer relationship management (CRM), human resource management (HRM), manufacturing management, and online retailers.
- Healthcare: Electronic Health Records (EHRs) are maintained in healthcare databases, ensuring patient data is accessible to medical professionals while maintaining confidentiality and compliance with privacy regulations.
- Finance: Banking systems rely on databases to manage customer accounts, transactions, and fraud detection. Financial institutions use databases for risk assessment, reporting, and compliance.
- Social Media: Social media platforms store vast amounts of user-generated content, profiles, and interactions in databases, allowing users to access and share information with others.
- Education: Educational institutions use databases to manage student records, grades, course schedules, and faculty information. This streamlines administrative processes and supports academic planning.
- Scientific disciplines: databases serve as foundational tools in various scientific endeavors, from gathering astronomical data to studying the human genome and

exploring protein properties in biochemistry, among numerous other scientific pursuits.

- Data mining and AI: Databases provide the foundation for data mining by offering structured storage, efficient management, and essential tools for data preparation and analysis that are widely used in data mining. Moreover, many machine learning models including deep learning rely on datasets, which are essentially databases, for training.

As discussed, data becomes valuable only when it is situated within a particular context. Metadata is the main tool for supplying context to data. ***Metadata** is data that provides information about other data, serving as a crucial layer of context and organization within information systems*. It describes the characteristics and attributes of data elements, such as their origin, format, relationships, and usage, facilitating efficient data management and retrieval. Metadata can include details like data creation dates, file sizes, author information, and data type specifications. In databases, metadata encompasses schema definitions, table structures, and indexing information, helping to define how data is stored, accessed, and manipulated. Figure 1d. illustrates the metada of the Student table.

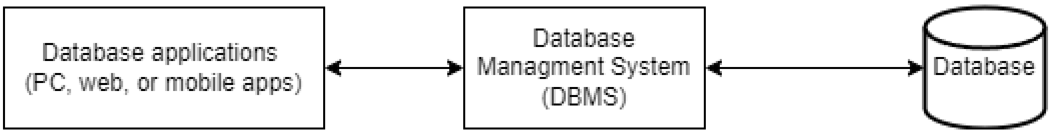


Figure 2. Components of a database system.

Databases exist in most computer software; however, they do not work in isolation. They are powered by a specialized type of software called **database management systems (DBMS)**, which are responsible for interacting with users and applications, managing data storage, enforcing data integrity, and providing various functionalities for data manipulation and retrieval. In short, *DBMSs are efficient tools that bridge the gap between data and information, allowing users to retrieve the desired data and gain valuable insights*. Examples of DBMS include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and MongoDB. *The database and the DBMS, along with the associated applications, are referred to as a* **database system,** as shown in Figure 2. Microsoft even provides Access, a software bundle that combines both database applications (having forms and reports) and DBMS within a single platform.

2. The importance of studying databases

As discussed in Section 1, databases are ubiquitous and play the cornerstone of many information systems. They are the location where the data are stored. Within this context, studying databases is essential for people with data or involved in information technology, especially for those who work as computer scientists or in the software engineering industry. It provides the foundational knowledge and skills needed to effectively manage, analyze, and leverage data to drive business success and innovation. Specially there are several reasons to study databases:

- Data Management: Databases are the backbone of modern data management systems. Understanding databases allows individuals to efficiently store, organize, and retrieve data, ensuring its integrity, consistency, and security.
- Information Retrieval: In today's data-driven world, the ability to retrieve information quickly and accurately is essential. Databases enable users to perform complex queries and retrieve relevant information from large datasets, supporting decision-making and analysis.
- Business Applications: Databases are widely used in business applications such as customer relationship management (CRM), enterprise resource planning (ERP), supply chain management (SCM), and financial systems. Knowledge of databases is essential for working with these systems and optimizing business processes.
- Data Analysis and Reporting: Databases serve as a foundation for data analysis and reporting. Understanding databases allows individuals to extract and manipulate data, perform statistical analysis, generate reports, and derive insights to support strategic decision-making.
- Software Development: Databases are integral to software development, particularly in web and application development. Knowledge of databases is essential for designing, developing, and maintaining database-driven applications, as well as for optimizing performance and scalability.
- Data Security and Privacy: Databases contain sensitive information that must be protected from unauthorized access, breaches, and misuse. Understanding databases enables individuals to implement security measures, such as access controls, encryption, and auditing, to safeguard data and ensure compliance with regulatory requirements.
- Career Opportunities: Proficiency in database technologies is highly sought after in various industries, including IT, finance, healthcare, marketing, and government. Learning about databases can open up a wide range of career opportunities, including database administration, data analysis, software development, and business intelligence.

3. Database users

The taxonomy of database users can vary depending on the context and the specific roles within an organization. However, broadly speaking, database users can be categorized into several groups based on their roles and responsibilities. Here is a general taxonomy of database users:

- **End Users**: End users are individuals who interact with applications or systems that rely on the database for data storage and retrieval. They may enter data into forms, view reports, or access information through web or mobile applications. End users include employees, customers, clients, and other users who interact with the organization's applications and services. They rarely interact directly with databases and DMBSs.
- **Database Administrators** (DBAs): DBAs are responsible for the overall management, administration, and maintenance of the database system. They perform tasks such as database installation, configuration, monitoring, backup and recovery, security management, and performance tuning. DBAs ensure the integrity, security, and availability of the database system.
- **Database Developers**: Database developers are responsible for designing, implementing, and maintaining the database schema and database applications. They create database tables, indexes, stored procedures, triggers, and other database objects. Database developers also write and optimize SQL queries, design data models, and ensure data consistency and integrity.
- **Database Architects**: Database architects are responsible for designing the overall architecture and infrastructure of the database system. They design data models, schemas, and database structures to meet the organization's requirements for scalability, performance, and reliability. Database architects also evaluate and select appropriate database technologies and platforms.
- **Application Developers**: Application developers are responsible for developing software applications that interact with the database system. They integrate database functionality into applications, write database queries, and implement database access logic. Application developers ensure that applications interact with the database securely, efficiently, and reliably.
- **Data Analysts**: Data analysts are responsible for analyzing and interpreting data stored in the database to derive insights and support decision-making. They write SQL queries, perform data analysis, create reports, and visualize data using tools such as BI (Business Intelligence) software. Data analysts help organizations understand trends, patterns, and correlations in their data.

4. History of Databases

The history of database systems is a fascinating journey that spans several decades. Here is an overview of the key milestones and developments in the evolution of databases:

a. 1950s-1960s - Early Databases:

During the late 1950s and early 1960s, the first electronic computers were being developed. With these computers came the need to store and manage large volumes of data efficiently.

Early databases were primarily hierarchical or network-based, where data was organized in tree-like or graph-like structures. Examples include IBM's Information Management System (IMS) and CODASYL databases.

b. 1970s - Relational Model

In 1970, Edgar F. Codd, an IBM researcher, introduced the relational model of data. This model represented data in tables with rows and columns, and it introduced the concept of structured query language (SQL) for querying and manipulating data.

The development of the relational database management system (RDBMS) began, with systems like IBM's System R and Oracle's early versions.

c. 1980s - Commercialization and Standardization

The 1980s saw the commercialization of relational database systems, with companies like Oracle, IBM, and Microsoft entering the market.

SQL became a standard query language for relational databases, and ANSI and ISO developed SQL standards.

d. 1990s - Client-Server Architecture and Object-Relational Databases:

The 1990s witnessed the adoption of client-server architectures, where databases were accessed remotely by client applications.

Object-oriented database systems emerged, attempting to combine the benefits of object-oriented programming with database capabilities. Examples include ObjectStore and Versant.

e. 2000s - Internet and Big Data:

The rise of the internet led to the development of web-based database applications, with databases serving as backends for websites and e-commerce platforms.

The term "Big Data" gained prominence as organizations started dealing with massive volumes of data, leading to the development of NoSQL databases and distributed data storage systems like Hadoop.

f. ***2010s - NoSQL and New Database Paradigms:***

NoSQL, standing for “Not Only SQL”, is a broad class of database management systems that diverge from the traditional relational database model. NoSQL databases, such as MongoDB, Cassandra, and Redis, gained popularity for their ability to handle unstructured and semi-structured data and provide horizontal scalability. They are designed to handle large volumes of data and are known for their flexibility, scalability, and performance.

- Flexibility: NoSQL databases do not require a fixed schema, allowing you to store and manage data in various formats like documents, key-value pairs, wide-column stores, or graphs.
- Scalability: They are built to scale out by distributing data across multiple servers, making them ideal for cloud computing and storage of massive amounts of data.
- Performance: With simpler data models and the ability to scale horizontally, NoSQL databases can provide faster responses to queries than traditional relational databases in certain scenarios.

NoSQL databases are categorized into four main types based upon their data models:

- Document Databases: Store data in documents similar to JSON or XML. These are useful for content management systems and e-commerce applications.
- Key-Value Stores: Simplest type of NoSQL databases that store data as a collection of key-value pairs. They are high performance for lookup queries.
- Column-Family Stores: Organize data into columns and are optimized for queries over large datasets, making them suitable for analyzing big data.
- Graph Databases: Designed to store and navigate relationships. They are powerful for social networking applications, fraud detection, and recommendation engines.

g. ***2020s and Beyond - Cloud Databases and AI Integration:***

The 2020s continued to see the rapid adoption of cloud-based databases, with major cloud providers offering Database as a Service (DBaaS) solutions.

Integration of artificial intelligence (AI) and machine learning (ML) into database systems became a prominent trend, enabling predictive analytics, automation, and data-driven decision-making.

Throughout this history, database systems have played a critical role in managing and organizing data for businesses, research, and various applications. They have evolved to handle diverse data types, support complex queries, ensure data integrity, and adapt to the changing needs of modern organizations in an increasingly data-centric world.

5. Current trends in databases

Database technology is continually evolving to meet the changing needs of organizations and the demands of modern applications. Several trends have emerged in recent years that are shaping the direction of database technology [1]:

a. Smaller and smaller database systems

Smaller and smaller database systems refer to database solutions that are designed to be lightweight, compact, and efficient, often catering to specific use cases with limited data storage and processing requirements. These systems are in contrast to large-scale, enterprise-level databases like Oracle, Microsoft SQL Server, or PostgreSQL. Here's an explanation of smaller and smaller database systems:

Embedded Databases

Embedded databases are compact database systems integrated directly into applications or software. They are used to manage data within the application itself, eliminating the need for a separate database server.

Common examples include SQLite, HSQLDB (HyperSQL Database), and Berkeley DB. These databases are often used in mobile apps, desktop applications, and embedded systems.

In-Memory Databases:

In-memory databases store data entirely in system memory (RAM), which allows for extremely fast data access and query performance.

They are suitable for scenarios where data needs to be processed quickly but can be lost in case of system failure. Redis and Memcached are popular in-memory databases used for caching and real-time data processing.

NoSQL Databases

Many NoSQL databases are designed to be smaller and more flexible than traditional relational databases. They excel at handling unstructured or semi-structured data and are often used in web and mobile applications.

Document-oriented databases like MongoDB and CouchDB, key-value stores like Redis and DynamoDB, and column-family stores like Cassandra are examples of smaller NoSQL databases.

Mobile Databases

Mobile databases are optimized for use in mobile applications where resources, including storage and processing power, are limited.

SQLite is commonly used in mobile development, offering a lightweight, self-contained, and serverless database engine.

Edge Databases

Edge databases are designed for edge computing scenarios where data processing and storage occur on local devices or edge servers rather than in a centralized data center.

They are used in IoT (Internet of Things) devices and scenarios where low latency and reduced bandwidth consumption are critical.

Single-User and Small-Team Databases

Some database systems are specifically designed for single users or small teams. These databases are often used in small businesses or for personal projects.

Examples include Microsoft Access and FileMaker, which are known for their ease of use and simplicity.

Containerized Databases

With the rise of containerization and micro services, smaller database systems are often packaged as lightweight containers, making them easy to deploy, manage, and scale in container orchestration platforms like Docker and Kubernetes.

Smaller and smaller database systems are valuable for their efficiency, agility, and suitability for specific tasks and environments where resource constraints or specialized requirements are a concern. They offer developers the flexibility to choose the right database solution for their particular use case while minimizing overhead and complexity.

b. Bigger and bigger database systems

Bigger and bigger database systems refer to database solutions designed to handle large volumes of data and complex workloads. These database systems are typically used by enterprises and organizations with extensive data storage and processing needs. Here's an explanation of bigger and bigger database systems:

Enterprise-Level Relational Databases

Large organizations often rely on enterprise-level relational database management systems (RDBMS) to manage their data. Examples include Oracle Database, Microsoft SQL Server, and IBM Db2.

These databases are known for their robustness, scalability, and support for advanced features such as high availability, disaster recovery, and advanced analytics.

Distributed Databases

Distributed databases are designed to distribute data across multiple servers or nodes to achieve horizontal scalability and fault tolerance.

Examples include Google Cloud Spanner, Amazon Aurora, and Apache Cassandra. They are commonly used for globally distributed applications and systems that require high availability.

Data Warehouses

Data warehouses are optimized for storing and analyzing large volumes of historical data. They are used for business intelligence, reporting, and data analytics.

Popular data warehousing solutions include Amazon Redshift, Google BigQuery, and Snowflake.

Blockchain and Distributed Ledger Databases

Blockchain and distributed ledger technologies are used to create secure, decentralized databases with immutability and transparency. They find applications in industries like finance, supply chain, and healthcare.

Multi-Model Databases

Some large-scale database systems are multi-model, supporting various data models (relational, document, graph, etc.) within a single database engine. This flexibility accommodates diverse data types and use cases.

Machine Learning and AI Integration:

Bigger database systems are increasingly incorporating machine learning and AI capabilities for advanced analytics, predictive modeling, and automated data management.

Hybrid and Multi-Cloud Deployments:

Enterprises are adopting hybrid and multi-cloud strategies, necessitating database systems that can operate seamlessly across multiple cloud providers and on-premises environments.

Bigger and bigger database systems are critical for handling the growing volume and complexity of data generated by modern organizations. They provide the scalability, performance, and features required to support mission-critical applications, data-driven decision-making, and the management of vast data repositories.

Time-Series Databases

Time-series databases are optimized for storing and querying time-stamped data, making them suitable for IoT, monitoring, and real-time analytics at scale.

InfluxDB and TimescaleDB are popular choices for time-series data.

Big Data and NoSQL Databases

As organizations deal with vast amounts of unstructured and semi-structured data, they turn to NoSQL databases designed for horizontal scalability and flexibility.

Big Data technologies like Hadoop and Spark, along with NoSQL databases like MongoDB, Cassandra, and Couchbase, are used for processing and analyzing massive data sets.

c. Multimedia Databases

Modern database system is required to integrate and to store various types of multimedia content, such as text, images, audio, video, and other forms of non-textual data. Multimedia databases are specialized databases designed to manage a wide variety of multimedia data types, including text, images, audio, video, and animation sequences. These databases are structured to handle the storage, retrieval, and organization of multimedia content, which is often large in size and complex in format. The multimedia databases have following key characteristics:

- **Data Types**: They support diverse data types such as text, graphics, images, animations, audio, and video. Each of these media types can be stored and retrieved efficiently.
- **Binary Storage**: Multimedia files are typically stored as binary strings and are encoded according to their specific file types to maintain integrity and quality.
- **Interrelated Data**: The data in multimedia databases are often interrelated, allowing for complex associations and queries that reflect the multifaceted nature of multimedia content.
- **Query and Retrieval**: These databases provide advanced query capabilities, enabling users to search for multimedia content based on various attributes and metadata.
- **Applications**: Multimedia databases are used in numerous applications, from digital libraries and educational platforms to entertainment and advertising, where the management of diverse media types is essential.

The main challenge for multimedia databases is to efficiently handle the large sizes and varied formats of multimedia data while providing fast and accurate retrieval mechanisms. They often incorporate specialized indexing and searching algorithms to meet these demands.

d. Client - Server and multi-tier database architecture

Client-Server and Multi-Tier Database Architectures are two common approaches to organizing and distributing the components of a database system. They help manage data access, improve scalability, and enhance the separation of concerns in database applications.

Client-Server Database Architecture

In a Client-Server Database Architecture, the database system is divided into two main components: the client and the server.

Client: The client component is responsible for presenting the user interface to the end-users and managing user interactions. It sends requests for data retrieval, updates, and other database operations to the server.

Server: The server component hosts the database itself and handles data storage, retrieval, and processing. It listens for incoming requests from clients, processes those requests, and returns the results to the clients.

Key characteristics of the Client-Server Database Architecture include:

- Centralized Data Management: The database resides on a dedicated server, which centralizes data storage and management. Clients access and manipulate data by sending requests to the server.
- Thin Clients: Clients in this architecture are often referred to as "thin clients" because they primarily focus on the user interface and user experience, while the majority of data processing occurs on the server.
- Scalability: The architecture allows for scalability by adding more server resources to handle increased client demands. This makes it suitable for applications with a large number of users.
- Improved Security: Security measures can be concentrated on the server, making it easier to implement access controls and encryption to protect sensitive data.

Multi-Tier Database Architecture

Multi-Tier Database Architecture, also known as N-Tier Architecture, extends the Client-Server model by adding multiple layers or tiers between the client and the database server. Each tier serves a specific purpose in processing data and managing application logic. The architecture typically consists of the following tiers:

Presentation Tier (Client): This tier represents the user interface and user interaction components, just like in the Client-Server model.

Application Tier (Middle Tier): The middle tier contains application servers responsible for processing business logic and application-specific functionality. It acts as an intermediary between the client and the data tier.

Data Tier (Database Server): The data tier contains the database server, where data is stored, managed, and retrieved. It may also include a separate database management system (DBMS).

Key characteristics of the Multi-Tier Database Architecture include:

- Separation of Concerns: The architecture separates the presentation, business logic, and data management into distinct tiers, making the application easier to develop, maintain, and scale.
- Improved Scalability and Load Balancing: Each tier can be scaled independently to handle varying levels of load. Load balancers can be used to distribute client requests across multiple application servers.
- Enhanced Security: Security measures can be applied at each tier, providing a more granular and layered approach to security. For example, authentication and authorization can be implemented in the application tier.

- Support for Different Technologies: Different tiers can be implemented using different technologies, enabling the use of specialized tools and languages for specific tasks.

To sum it up, the Client-Server Database Architecture consolidates data management while maintaining a distinct division between clients and servers. Conversely, the Multi-Tier Database Architecture introduces extra tiers, dividing responsibilities, improving scalability, and offering a more organized strategy for developing applications and handling data. The selection of these architectures relies on the particular needs and intricacies of the application.

Highlight & Summary

This chapter serves as a comprehensive exploration of the history and trends in database systems, offering valuable insights into the past, present, and future of data management. From the humble beginnings of data processing to the frontiers of distributed computing, the story of database systems is a testament to human innovation and the relentless pursuit of excellence in information technology.

Exercises

1. Distinguish following concepts: data, database, DBMS, and database systems.
2. Explain the types of user in databases.
3. What were the primary motivations behind the development of early database systems, and how did they differ from contemporary needs?
4. How did the evolution of database systems transition from early hierarchical and network models to the dominance of relational databases?
5. Describe several trends in the development of the database system
6. Name one advantage of NoSQL database system compare to the SQL system.

Chapter 2: The Relational Model

In the continuously changing field of data management, the relational database model is still the most prevalent database model today. Created by Edgar F. Codd in 1970 [2], the relational model has evolved into the foundation of contemporary database systems. It offers a strong framework for the organization and manipulation of structured data. This chapter presents the essential concepts and principles that form the basis of the relational model.

By the end of this chapter, readers should achieve the following goals:

- Able to explain the data model concepts and list important data models in history.
- Understand the philosophy of the relational data model.
- Able to describe the components of the relation data model: relation, attributes, tuples, schemas, domains, relation instance, keys of relations.
- Understand and able to use the basic operation on the relation data model: union, intersection, difference, projection, selection, renaming, products, theta join, and natural join.

1. Data Models in the history of Databases

A data model is a conceptual framework that defines how data is organized and structured in a database. It provides a way to represent and organize data, making it easier to understand, manage, and manipulate within a database system. Data models serve as a bridge between the real-world entities and the physical storage of data in a database, allowing for efficient storage, retrieval, and management of information. A data model often define 3 following aspects:

- Structure of data: how to store and organize data.
- Operations on data: the details of actions we work with data.
- Constraints of data: the rules that data must follow to ensure the correctness.

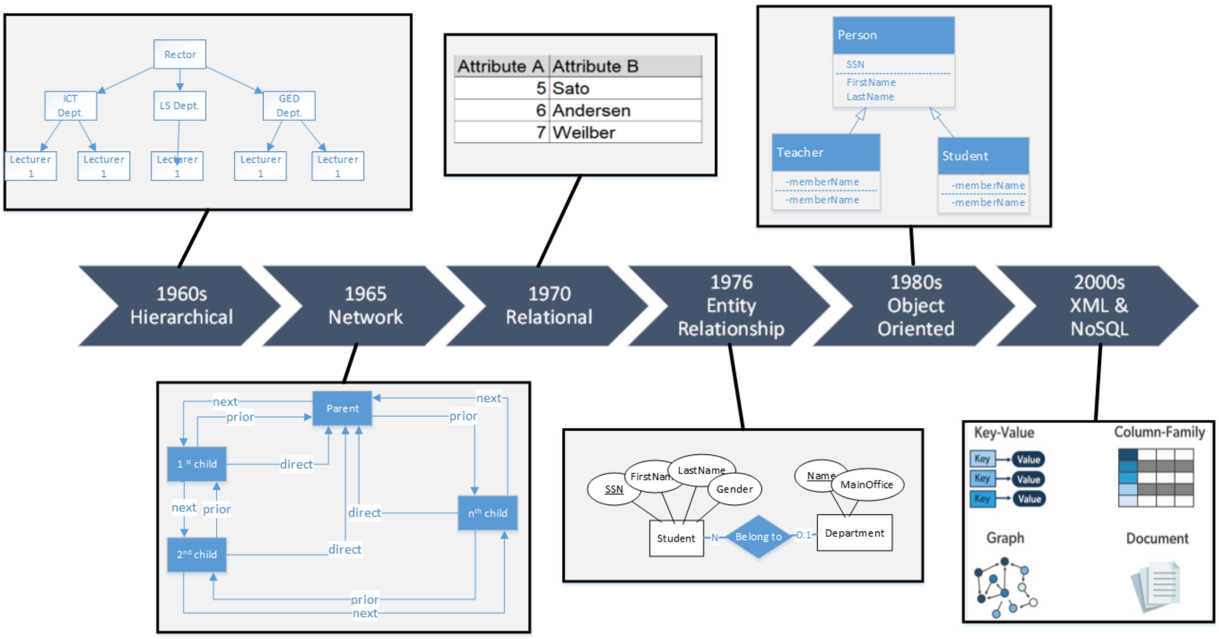


Figure 3. Notable data models in history.

Through the evolution of databases, different data models were invented and can be traced through folowing key milestones (illustrated by Figure 3) [3]:

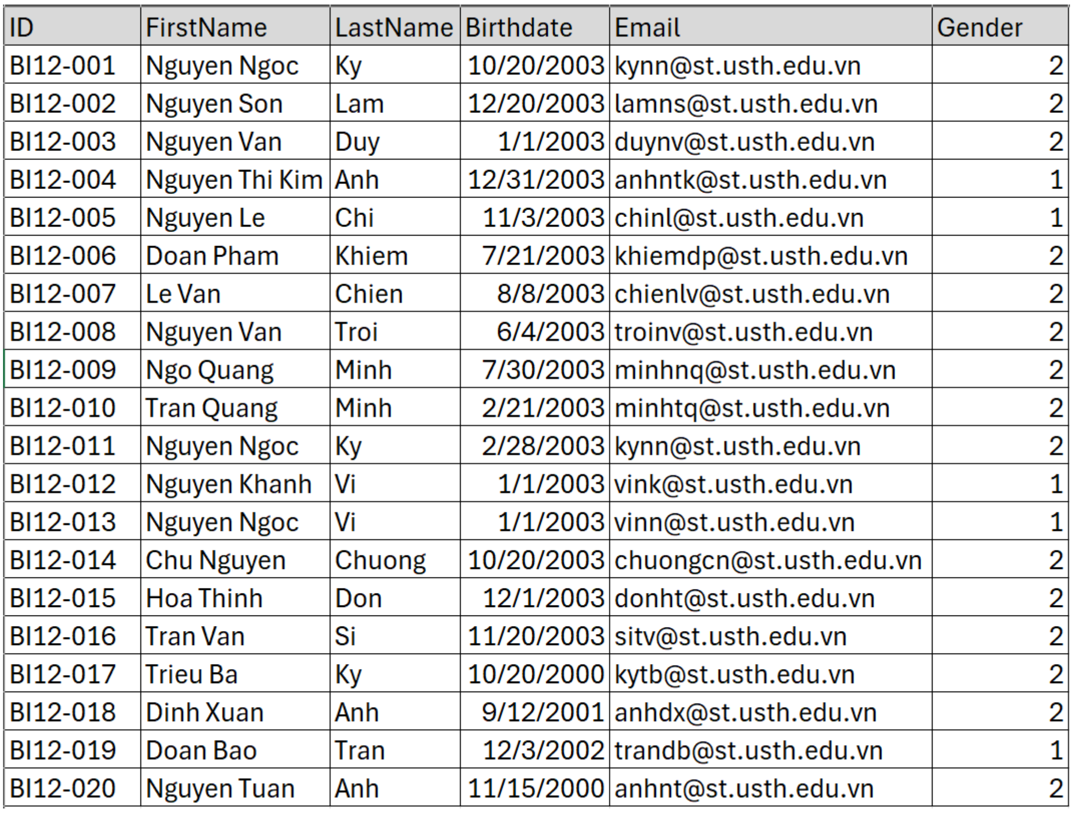
- Hierarchical and Network Models (1960s): Before the relational model emerged, hierarchical and network data models were prevalent. The Hierarchical Model organized data in a tree-like structure, and the network model allowed for more complex relationships. However, these models had limitations in representing certain types of relationships and were often complex to implement and maintain. The Network Model was introduced in 1965 by the Conference on Data Systems Languages (CODASYL). This model allowed more complex relationships by enabling records to have multiple parent and child records. It provided greater flexibility than the hierarchical model but was also more complex to implement and manage.

- Relational Model (1970): Edgar F. Codd introduced the relational model in 1970, revolutionizing the field of database management. The relational model organized data into tables with rows and columns, emphasizing simplicity and mathematical foundations. This model introduced the concepts of primary keys, foreign keys, and normalization, providing a more flexible and intuitive way to represent relationships between entities.
- Entity-Relationship Model (1976): Peter Chen proposed the Entity-Relationship (ER) model as a graphical representation for database design. This model introduced entities, attributes, and relationships in a visual format, making it easier for designers to conceptualize and communicate database structures.
- Object-Oriented Models (1980s): With the rise of object-oriented programming, object-oriented data models emerged. These models extended the concepts of the relational model to include objects, classes, and inheritance, allowing for a more natural representation of complex data structures. ObjectStore, Versant, and db4o are some examples of object oriented databases.
- XML and NoSQL Models (2000s): As the internet and web applications gained prominence, XML (eXtensible Markup Language) and NoSQL (Not Only SQL) databases emerged. XML provided a flexible way to represent hierarchical data, and NoSQL databases offered alternatives to traditional relational databases, accommodating large-scale and unstructured data. There are some main types of NoSQL databases:
    - Key-Value Stores: Simple data storage with a key and a value (e.g., Redis, DynamoDB).
    - Document Stores: Store data in document format (e.g., JSON, BSON) (e.g., MongoDB, CouchDB).
    - Column-Family Stores: Store data in columns rather than rows (e.g., Apache Cassandra, HBase).
    - Graph Databases: Use graph structures with nodes, edges, and properties to represent and store data (e.g., Neo4j, OrientDB). Recently, graph databases gained popularity, especially in applications involving complex relationships and network structures. Graph databases are optimized for efficiently traversing and querying graph-like structures.

Throughout this history, the field of data modeling has evolved to address the changing needs of data management in various domains. The relational model remains a foundational concept; however, the diversity of data models reflects the diverse nature of data and the requirements of different applications and industries. Today, the data modeling landscape continues to adapt to emerging technologies, such as Big Data, machine learning, and distributed computing. Interestingly, many machine learning models and data mining techniques still use data in the tabular format, which is a simple relational database.

2. Basics of the Relational Model

The Relational Model was first proposed by Edgar F. Codd in 1970 and has since become the foundation for most modern database management systems. The relational model organizes data into tabular structure, where each table consists of rows and columns. In this section, we will present important concepts of the relational models and exemplify them using Student relation as shown in Figure 4.



ID	FirstName	LastName	Birthdate	Email	Gender
BI12-001	Nguyen Ngoc	Ky	10/20/2003	kynn@st.usth.edu.vn	2
BI12-002	Nguyen Son	Lam	12/20/2003	lamns@st.usth.edu.vn	2
BI12-003	Nguyen Van	Duy	1/1/2003	duynv@st.usth.edu.vn	2
BI12-004	Nguyen Thi Kim	Anh	12/31/2003	anhntk@st.usth.edu.vn	1
BI12-005	Nguyen Le	Chi	11/3/2003	chinl@st.usth.edu.vn	1
BI12-006	Doan Pham	Khiem	7/21/2003	khiemdp@st.usth.edu.vn	2
BI12-007	Le Van	Chien	8/8/2003	chienlv@st.usth.edu.vn	2
BI12-008	Nguyen Van	Troi	6/4/2003	troinv@st.usth.edu.vn	2
BI12-009	Ngo Quang	Minh	7/30/2003	minhnq@st.usth.edu.vn	2
BI12-010	Tran Quang	Minh	2/21/2003	minhtq@st.usth.edu.vn	2
BI12-011	Nguyen Ngoc	Ky	2/28/2003	kynn@st.usth.edu.vn	2
BI12-012	Nguyen Khanh	Vi	1/1/2003	vink@st.usth.edu.vn	1
BI12-013	Nguyen Ngoc	Vi	1/1/2003	vinn@st.usth.edu.vn	1
BI12-014	Chu Nguyen	Chuong	10/20/2003	chuongcn@st.usth.edu.vn	2
BI12-015	Hoa Thinh	Don	12/1/2003	donht@st.usth.edu.vn	2
BI12-016	Tran Van	Si	11/20/2003	sitv@st.usth.edu.vn	2
BI12-017	Trieu Ba	Ky	10/20/2000	kytb@st.usth.edu.vn	2
BI12-018	Dinh Xuan	Anh	9/12/2001	anhdx@st.usth.edu.vn	2
BI12-019	Doan Bao	Tran	12/3/2002	trandb@st.usth.edu.vn	1
BI12-020	Nguyen Tuan	Anh	11/15/2000	anhnt@st.usth.edu.vn	2

Figure 4. The student relation

a. Relation

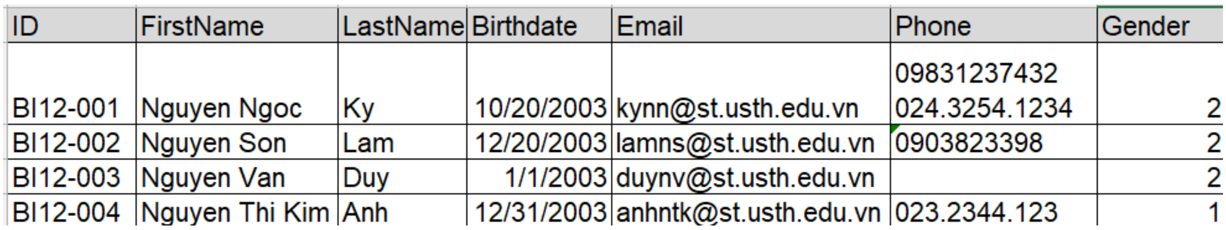
A relation, a.k.a. table, represent a set of specific entities, such as students, customers, products, or orders. Data of a relation is organized into rows and columns (reffered to as tupples and attributes, detailed in the next subsections). This terminology comes from relational algebra (see section 5), a branch of mathematics that deals with sets and relations. These phrases (realation, attribute, tuple) were coined by academics and are not commonly used in the software industry.

A relation must satisfy following conditions:

- Atomicity: Each attribute value in a tuple is atomic, meaning it is indivisible. This ensures that data is stored in its simplest form.

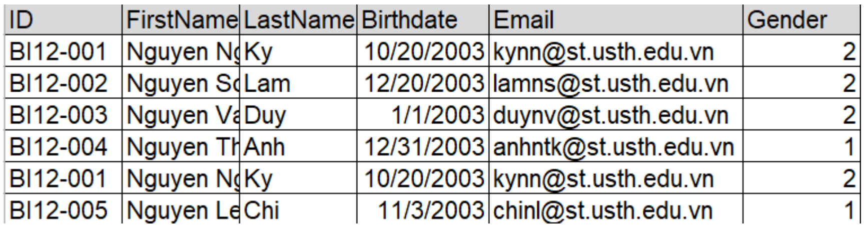
- Uniqueness: The tuples in a relation are unique. No two tuples can have exactly the same values for all attributes, which is often enforced by defining a primary key.
- Order Independence: The order of tuples in a relation does not matter, nor does the order of attributes in a tuple.

Figure 5 and 6 demonstrate 02 data tables found in reality, yet they are not relations.



ID	FirstName	LastName	Birthdate	Email	Phone	Gender
BI12-001	Nguyen Ngoc	Ky	10/20/2003	kynn@st.usth.edu.vn	09831237432 024.3254.1234	2
BI12-002	Nguyen Son	Lam	12/20/2003	lamns@st.usth.edu.vn	0903823398	2
BI12-003	Nguyen Van	Duy	1/1/2003	duynv@st.usth.edu.vn		2
BI12-004	Nguyen Thi Kim	Anh	12/31/2003	anhntk@st.usth.edu.vn	023.2344.123	1

Figure 5. Non-relation: possible multiple entries in Phone.



ID	FirstName	LastName	Birthdate	Email	Gender
BI12-001	Nguyen N	Ky	10/20/2003	kynn@st.usth.edu.vn	2
BI12-002	Nguyen S	Lam	12/20/2003	lamns@st.usth.edu.vn	2
BI12-003	Nguyen V	Duy	1/1/2003	duynv@st.usth.edu.vn	2
BI12-004	Nguyen T	Anh	12/31/2003	anhntk@st.usth.edu.vn	1
BI12-001	Nguyen N	Ky	10/20/2003	kynn@st.usth.edu.vn	2
BI12-005	Nguyen L	Chi	11/3/2003	chinl@st.usth.edu.vn	1

Figure 6. Non-relation: duplicated rows.

b. Relation instance

The Student relation is dynamic and can change over time. We can add a new student to the relation, edit/modify information related to a specific student, or delete a record of the student from the relation.

While alterations to the schema of a relation are less frequent, there are instances where we may need to add or remove attributes. However, modifying the schema, though feasible in commercial database systems, can be a costly process. This is because each of the potentially millions of tuples must be rewritten to incorporate or eliminate components. We will refer to a collection of tuples belonging to a specific relation as an instance of that relation.

In Figure 4, we have an instance of a relation referred to as "Student." Each row corresponds to a specific student, and each column corresponds to a characteristic of students.

c. Attribute

Attributes assign names to the columns of a relation; in Figure 4, these attributes are ID, FirstName, LastName, Birthdate, Email and Gender. The attributes are situated at the column headers. Typically, an attribute elucidates the significance of entries within the column beneath it. For example, the column labeled ID, an attribute, contains the student ID of each student in the table.

d. Functional Dependency (FD)

Functional dependency (FD) is a constraint between two sets of attributes in a relation from a database. Specifically, a functional dependency, denoted as X→Y indicates that if two tuples (rows) of a relation have the same value for attribute X, they must also have the same value for attribute Y. In this context, X is called the determinant, and Y is the dependent.

A functional dependency X→Y is trivial if Y is a subset of X. For example, {A, B}→A is a trivial dependency because A is a subset of {A,B}. A functional dependency X→Y is completely non-trivial if X and Y have no attributes in common.

In Student relation there are some completely non-trivial FDs:

- {ID} → {FirstName, LastName, BirthDate, Email, Gender}
- {Email} → {FirstName, LastName, ID, Email, Gender}

FDs are used when we find key of a relation (see section 3b of this chapter) and to identify potential anomalies in the database design and guide the normalization process (see section 5c of chapter 5).

e. Tuple

The rows in a relation, excluding the header row that contains attribute names, are known as tuples. Each tuple possesses a component for every attribute of the relation. For instance, the initial tuple in Figure 4 comprises six elements: BI12-001, Nguyen Ngoc, Ky, 2003-10-20, kynn@st.usth.edu.vn and 2, corresponding to the attributes ID, FirstName, LastName, Birthdate, Email and Gender, respectively. When representing a tuple independently, not as part of a relation, we typically employ commas to separate the components, and parentheses to enclose the tuple. For instance, the first tuple in Figure 4 is expressed this way.

```
(BI12-001, Nguyen Ngoc, Ky, 2003-10-20, kynn@st.usth.edu.vn, 2)
```

It is important to note that when a tuple is presented independently, the attributes are omitted, necessitating some indication of the associated relation. We consistently adhere to the order in which the attributes were initially listed in the relation schema.

f. Schema

As discussed, an instance of a relation can be considered as its snapshot, meaning that its data can change over time. However, the structure of that relation should be rather stable and it presents the logical design of the relation. We describe the schema by listing the relation name followed by a set of attributes enclosed in parentheses. Consequently, the schema for the Student relation in Figure 4 is displayed as follows:

```
Student(ID, FirstName, LastName, Birthdate, Email, Gender)
```

Within the relational model, a database comprises one or more relations. The collection of schemas for these relations within a database is referred to as a relational database schema, alternatively known simply as a database schema.

3. Data integrity

Data integrity refers to the accuracy, consistency, and reliability of data over its lifecycle. In the context of relational algebra, data integrity involves a set of rules and constraints that maintain the correctness and validity of the data within a relational database. This section explores the various mechanisms by which relational algebra enforces data integrity, emphasizing its importance and implementation.

There are several types of data integrity in relational databases, each serving to maintain different aspects of data correctness.

a. Domain Integrity

The relational model mandates that every element within each tuple must be atomic, meaning it should be a basic type like integer or string. It is not allowed for a value to be a complex structure like a record, set, list, array, or any other type that can reasonably be divided into smaller components.

It is also assumed that every attribute in a relation is associated with a specific domain, indicating a particular elementary type. Each tuple's components in the relation must possess values within the domain of the corresponding column. For instance, in the Student relation shown in Figure 4, the first five components of tuples must be a string, and the last one must be an integer. For more rigid domain, the domain for value of Gender could be limited in {1, 2, 3} with 1 for male, 2 for female, and 3 for other.

To include the domain or data type for each attribute in a relation schema, we can achieve this by appending a colon and specifying the type after each attribute. As an illustration, we could express the schema for the Student relation as follows:

```
Student (ID: string, FirstName: string, LastName: string, Birthdate: Datetime,
Email: string, Gender: integer)
```

b. Entity Integrity by Keys

The entity Integrity ensures that each entity (row) in a database is uniquely identifiable. This is typically achieved using the **key** constraint. It is a set of attributes of a relation with two criteria:

- **Uniqueness**: a key must ensure that no two rows in the table have the same value for the key attribute or combination of attributes. This uniqueness is crucial because it allows the key to reliably distinguish one record from another within the table. Each value in the key attribute(s) must be unique across all rows, preventing duplicate entries.
- **Minimal**: the attribute is minimal in the sense that we cannot exclude any attribute so that the new set still satisfy the uniqueness property. So the set must be able to uniquely identifying a tuple in the relation with smallest possible number of attributes

Consider the relation `Student (ID, FirstName, LastName, Birthdate, Email, Gender).` Both {ID} and {Email} can serve as the keys because the table has unique values at ID and Email while these sets consist of only one attribute so they are minimal. We call all the sets satisfying two mentioned criteria **“candidate keys”.** Among them, we opt for a main set to use in practice which is called **primary key,** then the other candidate keys is referred to as **“alternate keys”.** We denote the attribute or attributes forming a key for a relation by underlining the key attribute(s). As an illustration, the schema of the Student relation might be represented as:

```
Student (ID, FirstName, LastName, Birthdate, Email, Gender).
```

A key consisting of multiple attributes is called **composite key** or **compound key**. For example, the relation `Assessment` expressing the assessment result of a student taking the test of a subject at a time (1st, 2nd, …) has the compound key as below:

```
Assessment (StudentID, SubjectID, Time, Mark)
```

The **superkey** concept, on the other hand, refers to the super sets of a key of a relation. In the Student table, some examples of its superkeys are `{ID}, {ID, FirstName}, {ID, FirstName, LastName, Birthdate, Email, Gender}, {Email}, {BirthDate, Email}.`

In some situations, the actual data of a relation are not really suitable to find a good key since natural keys are either not available, not stable, or not efficient for use as primary keys. The surrogate key is the solution for such problem. **Surrogate Key** is a type of unique identifier used in a database table to uniquely identify each record. Unlike natural keys, which are derived from the actual data, surrogate keys are artificial and are typically generated by the database system itself. The bellow examples illustrate the use of surrogate keys:

- Relation `Teacher (FirstName, LastName, SSN, BirthDate, Email, Rank, Affiliation, DepartmentID)` has SSN (Social Secure Number) attribute as a candidate key; however, its data is too complex to be the primary key. A surrogate key `ID` with integer data type can be use as the primary key of this relation, resulting `Teacher (ID, FirstName, LastName, SSN, BirthDate, Email, Rank, Affiliation, DepartmentID).`
- Relation `Subject` has two natural atributes `Name` and `NoCredits` where Name is unique among subjects, yet the `Name` is in text format which is not convenient in sorting, searching, and matching operations. A design of relation `Subject (ID, Name, NoCredits)` with the surrogate key ID is commonly used in practice.

c. *Referential Integrity by Foreign Keys*

A foreign key is an attribute or a set of attributes in one relation that refers to the primary key in another relation. By linking relation together, foreign keys help ensure that the data remains consistent and that relationships between entities are properly enforced.

A foreign key is defined by the following characteristics:

- Referential Integrity: The values in the foreign key column(s) must match the values in the referenced primary key column(s) of the parent table or be null if null values are allowed.
- Establishes Relationships: Foreign keys establish a relationship between two tables, typically representing a parent-child relationship where the foreign key table (child) references the primary key table (parent).

Consider two tables:

- `Student (ID, FirstName, LastName, Birthdate, Email, Gender).`
- `Assessment (StudentID, SubjectID, Date, Mark)`

The `StudentID` of table `Assessment` (denoted as `Assessment.StudentID`) refers to the primary key `ID` of table `Student (Student.ID),` so it is a foreign key, meaning that the value of StudentID must match `ID.`

d. Null constraint

Null constraint is a rule that specifies whether a column in a table can hold a null value. Null values represent the absence of data, which can be different from empty strings or zeroes, depending on the context. Managing null values effectively is crucial for maintaining data integrity and ensuring accurate query results. This section explores the concept of null constraints, their significance, and best practices for handling null values in relational databases.

A null value in a relational database indicates that the data for a particular column in a row is missing or unknown. It is important to distinguish null values from other forms of data absence, such as empty strings for text fields or zeroes for numeric fields. Null values are useful for representing missing or inapplicable information without making assumptions about the data. A null constraint determines whether a column can accept null values. There are two main types of null constraints:

- NOT NULL Constraint: Specifies that a column must always have a value and cannot be null.
- Allowing Nulls: By default, columns in a table can accept null values unless explicitly constrained otherwise.

Consider the `Students (ID, FirstName, LastName, Birthdate, Email, Gender)` relation. The attribute Email is allowed to be NULL, meaning that emails of some students can be empty. The other attribute must be NOT NUL.

e. User-Defined Integrity

User-defined integrity constraints enforce specific business rules that ensure the validity of the data according to the business logic. This type of integrity can sometimes be referred as CHECK constraint since many DBMS use the CHECK keyword when defining the table.

For example, a business rule states that the Student relations have following constraints:

- The email attribute of Student must always be in format '_%@_%._%' where _ represents a single character and % represents zero or more characters.
- Gender is not only in the integer data type, the values must be in one of the following: 1 (male), 2 (female), 3 (other).

Other example of user-define integrity is that a student cannot take the test of a subject more than 10 times.

4. Database Schema

Previous sections present how to describe the structure of relations. However, it is essential to describe the whole database. A **database schema** is a blueprint or architecture of how a database is structured. In the context of relational databases, it defines the relations, fields, relationships and other elements (such as views and indices, see chapter 4) within the database. Essentially, it outlines how data is organized and how the relationships among data are managed. The following subsections provide database schema examples.

a. Database Schema of a simplified Academic Management System

Section 2 introduces the Student relation solely. We now consider other relations of a database for a simplified Academic Management System (referred to as Lite AMS) which is collection of data designed to manage the administrative and academic functions of an educational institution such as a university. For the sake of simplicity, this database consist of data for only few core businesses such as efficient management of student records, teachers, subjects, classes. It provides a centralized repository for storing and retrieving information related to enrollment, class registration, and teaching allocation. The full version of database for AMS will be discussed at chapter 5. The schema off Lite AMS is below:

- Department (ID, Name, MainOffice, DirectorID)
- Teacher (ID, FirstName, LastName, SSN, BirthDate, Email, Rank, Affiliation, DepartmentID)
- Student(ID, FirstName, LastName, Birthdate, Email, Gender, DepartmentID)
- AcademicYear (ID, Name, StartingDate, EndingDate)
- Subject (ID, Name, NoCredits)
- Class (Name, SubjectID, AcademicYearID)
- ClassStudent (StudentID, ClassID)
- TeachingRole (ID, RoleName)
- ClassTeacher (ClassID, TeacherID, TeachingRoleID)

And here are their relationships:

Relationship	Cardinality	Foreign key
Student - Department	One - many	Student.DepartmentID refers to Department.ID
Teacher - Department	One - many	Teacher.DepartmentID refers to Department.ID
Department – Teacher	One – Many	Department.DirectorID refers to Teacher.ID
Subject - Class	One-many	Class.SubjectID refers to Subject.ID
Class - AcademicYear	One-many	Class.AcademicYearID refers to AcademicYear.ID
Class – Student	Many-Many	One student follows multiple classes and a class consists of multiple students. This many many relationship is express by the intermediate relation ClassStudent. This relation has 02 foreign keys: ClassStudent.StudentID refers to Student.ID and ClassStudent.ClassID refers to Class.ID.
Class – Teacher	Many-many	One class can be taught by some teachers, a teacher can teach multiple class. This many many relationship is express by the intermediate relation ClassTeacher. ClassTeacher.ClassID

Relationship	Cardinality	Foreign key
		is the foreign key refers to Class.ID and ClassTeacher.TeacherID refers to Teacher.ID
Student - Subject	Many-many	One student has assessment results of multiple subjects, while a subject has assessment resujlts for different students. Noteably, a student can take the test of a subject several times. Assessment.StudentID refers to Student.ID and Assessment.SubjectID refers to Subject.ID

b. *Database Schema of Northwind*

The Northwind database is a sample database that was originally created by Microsoft for demonstrating the capabilities their DBMS products including SQL Server and Access, then it was later ported to various formats including MySQL, PostgreSQL. It has been used as the basis for tutorials and examples, providing a rich dataset for learning SQL, database design, and data analysis.

The Northwind database simulates a fictional company called Northwind Traders, which imports and exports specialty foods from around the world. It consists of following relations:

- Customers (CustomerID, CustomerName, ContactName, Address, City, PostalCode, Country)
- Employes (EmployeeID, LastName, FirstName, BirthDate, Photo, Notes)
- Shippers (ShipperID, ShipperName, Phone)
- Categories (CategoryID, CategoryName, Description)
- Suppliers (SupplierID, SupplierName, ContactName, Address, City, PostalCode, Country, Phone)
- Products (ProductID, ProductName, SupplierID, CategoryID, Unit, Price)
- Orders (OrderID, CustomerID, EmployeeID, OrderDate, ShipperID)
- OrderDetails (OrderDetailID, OrderID, ProductID, Quantity)

Where Orders relation holds records of customer orders, including order dates, shipping dates, and shipping methods. The OrderDetails relation expresses the information on the specific products included in each order, including quantities and prices. The Suppliers expreses the information about the companies that supply products to Northwind Traders. The Shippers keeps track with the list of companies that handle shipping for Northwind Traders.

There are some one-many relationships:

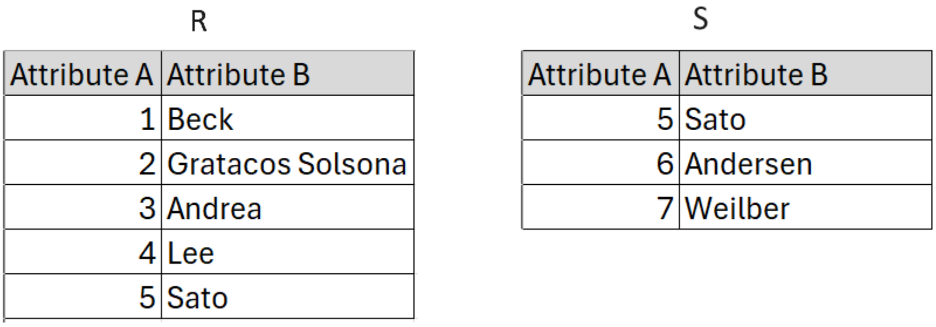
- Products.SupplierID refers to Suppliers.SupplierID
- Products.CategoryID refers to Categories. CategoryID
- Order.CustomerID refers to Customers.CustomerID
- Order.EmployeeID refers to Employees.EmployeeID
- Order.ShipperID refers to Products.SupplierID
- OrderDetails.OrderID refers to Orders.OrderID
- OrderDetails.ProductID refers to Products.ProductsID

5. Relational Algebra

In the relational data model, data is organized into tables, where each table consists of rows and columns. Operations on relational data are essential for querying and manipulating the information stored in these tables. Relational algebra is a mathematical framework and a formal query language used to describe operations on relational databases. It provides a set of operations that can be applied to relations (tables) to manipulate and retrieve data. The relational algebra serves as the theoretical foundation for the design and implementation of RDBMSs. In this section, we will take a look on some fundamental operations on relational data.

a. Set operation on Relations

Set operations in the relational data model involve manipulating and combining sets of tuples (rows) from different relations (tables). These operations are analogous to set operations in mathematical set theory and are essential for retrieving, comparing, and combining data in a relational database. The primary set operations in the relational data model include **union**, **intersection**, and **difference**. To demonstrate the set operation on Relations, in this section we examine the two relations R and S as shown in the Figure 7.



R

Attribute A	Attribute B
1	Beck
2	Gratacos Solsona
3	Andrea
4	Lee
5	Sato

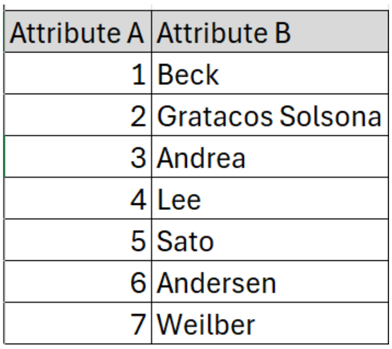
S

Attribute A	Attribute B
5	Sato
6	Andersen
7	Weilber

Figure 7. R and S relations.

Union

The union operation combines the tuples from two relations and eliminates duplicates to produce a new relation. The resulting relation contains all unique tuples that are present in either of the original relations. In mathematical terms, if R and S are two relations, then $R \cup S$ represents their union, as demonstrated in Figure 8.

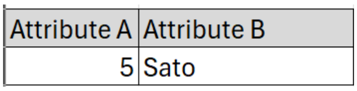


Attribute A	Attribute B
1	Beck
2	Gratacos Solsona
3	Andrea
4	Lee
5	Sato
6	Andersen
7	Weilber

Figure 8. Union of R and S.

Intersection

The intersection operation returns a relation containing only the tuples that are common to both input relations. In mathematical terms, if R and S are two relations, then $R \cap S$ represents their intersection as in Figure 9.

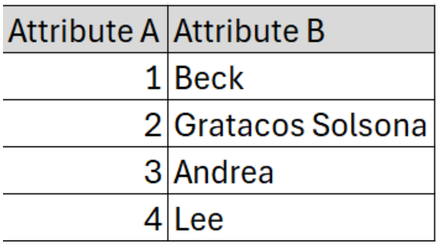


Attribute A	Attribute B
5	Sato

Figure 9. Intersection of R and S.

Difference

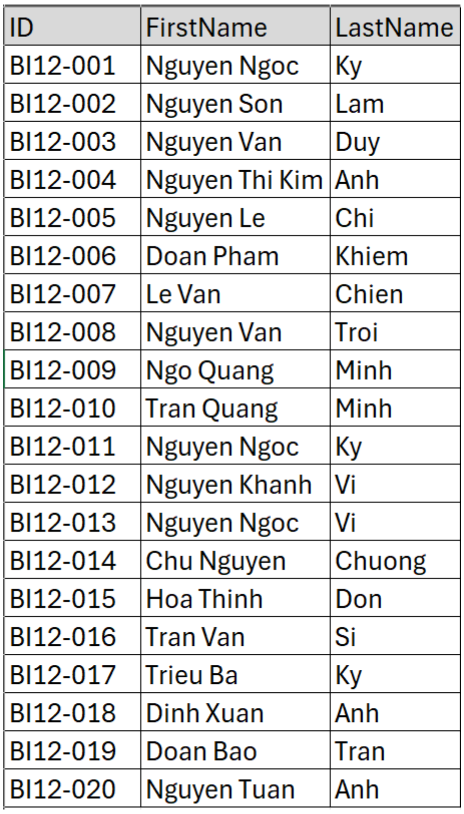
The difference operation returns a relation containing tuples that are present in the first relation but not in the second relation. In mathematical terms, if R and S are two relations, then $R - S$ represents their difference. Please noted that $R - S$ is different from $S\ with\ R$ (see Figure 10).



Attribute A	Attribute B
1	Beck
2	Gratacos Solsona
3	Andrea
4	Lee

Figure 10. R-S as an example of difference.

b. Projection



ID	FirstName	LastName
BI12-001	Nguyen Ngoc	Ky
BI12-002	Nguyen Son	Lam
BI12-003	Nguyen Van	Duy
BI12-004	Nguyen Thi Kim	Anh
BI12-005	Nguyen Le	Chi
BI12-006	Doan Pham	Khiem
BI12-007	Le Van	Chien
BI12-008	Nguyen Van	Troi
BI12-009	Ngo Quang	Minh
BI12-010	Tran Quang	Minh
BI12-011	Nguyen Ngoc	Ky
BI12-012	Nguyen Khanh	Vi
BI12-013	Nguyen Ngoc	Vi
BI12-014	Chu Nguyen	Chuong
BI12-015	Hoa Thinh	Don
BI12-016	Tran Van	Si
BI12-017	Trieu Ba	Ky
BI12-018	Dinh Xuan	Anh
BI12-019	Doan Bao	Tran
BI12-020	Nguyen Tuan	Anh

Figure 11. The result of $\pi_{\{ID,FirstName,LastName\}}(Student)$.

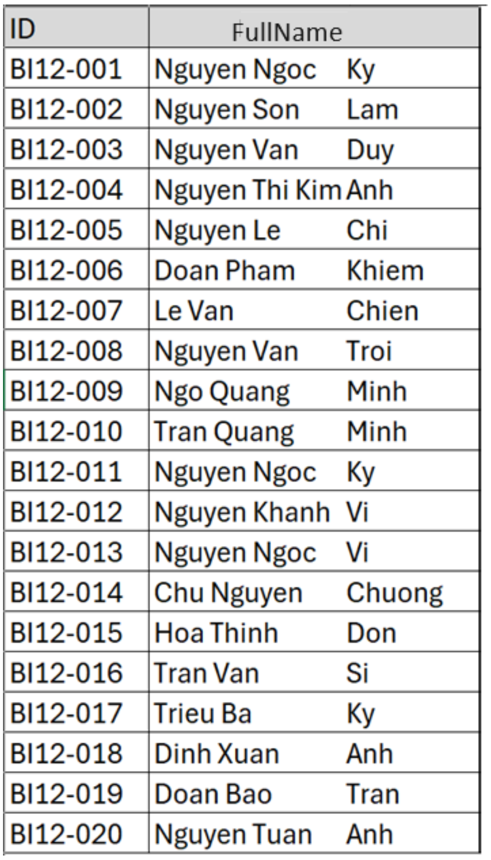
Projection is one of the fundamental operations in the relational data model. It allows you to create a new relation that includes only certain columns (attributes) from an existing relation. In other words, projection extracts specific attributes from a relation while discarding the others. The result is a subset of the original relation with a reduced number of columns.

In mathematical terms, if R is a relation with attributes $A_1, A_2, \dots A_n$ then the projection of R on attributes $B_1, B_2, \dots B_n$ is denoted as $\pi_{\{B_1,B_2,\dots B_m\}}(R)$. The result is a new relation with only the selected attributes. For example, the result $\pi_{\{ID,FirstName,LastName\}}(Student)$ is shown in Figure 11.

The project operator can be extended to manipulate existing data of attributes. For instance, we can concatenate FirstName and LastName attributes to result in FullName in the output relation: $\pi_{\{ID,FirstName\ +'\ '+\ LastName\}}(Student)$. The results is shown at Figure 12.

Key points about projection in the relational model:

- Column Selection: Projection allows you to choose and include only the attributes that are relevant to a specific query or analysis. It helps in reducing the amount of data retrieved from the database, making queries more efficient.
- Elimination of Duplicates: The result of a projection operation typically removes duplicate tuples, ensuring that the resulting relation contains only distinct combinations of the selected attributes.
- Schema of Result: The resulting relation from a projection has a new schema that includes only the selected attributes, and it inherits the data types and constraints of those attributes.



ID	FullName
BI12-001	Nguyen Ngoc Ky
BI12-002	Nguyen Son Lam
BI12-003	Nguyen Van Duy
BI12-004	Nguyen Thi Kim Anh
BI12-005	Nguyen Le Chi
BI12-006	Doan Pham Khiem
BI12-007	Le Van Chien
BI12-008	Nguyen Van Troi
BI12-009	Ngo Quang Minh
BI12-010	Tran Quang Minh
BI12-011	Nguyen Ngoc Ky
BI12-012	Nguyen Khanh Vi
BI12-013	Nguyen Ngoc Vi
BI12-014	Chu Nguyen Chuong
BI12-015	Hoa Thinh Don
BI12-016	Tran Van Si
BI12-017	Trieu Ba Ky
BI12-018	Dinh Xuan Anh
BI12-019	Doan Bao Tran
BI12-020	Nguyen Tuan Anh

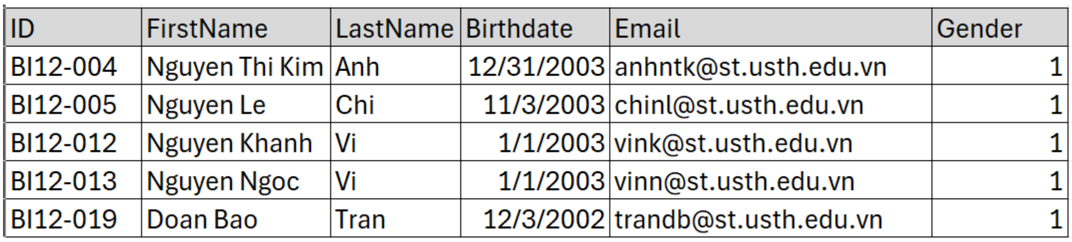
Figure 12. The result of extended projection $\pi_{\{ID, FirstName\ +'\ '+\ LastName\}}(Student)$*.*

c. *Selection*

Selection is a fundamental operation in the relational data model that allows you to retrieve a subset of rows (tuples) from a relation (table) based on a specified condition. In other words, selection filters the rows that meet a particular criterion, and the result is a new relation containing only those rows that satisfy the given condition.

In mathematical terms, if R is a relation and C is a condition, then the selection of R based on C is denoted as $\sigma_C(R)$. The condition C is expressed as a logical expression involving attributes of the relation, and only the rows that satisfy this condition are included in the result.

In this example, the condition is "`Gender = 1`" and the result of the query $\sigma_{Gender=1}(Student)$ is a new relation containing only the female students in the Student relation (Figure 13).



ID	FirstName	LastName	Birthdate	Email	Gender
BI12-004	Nguyen Thi Kim	Anh	12/31/2003	anhntk@st.usth.edu.vn	1
BI12-005	Nguyen Le	Chi	11/3/2003	chinl@st.usth.edu.vn	1
BI12-012	Nguyen Khanh	Vi	1/1/2003	vink@st.usth.edu.vn	1
BI12-013	Nguyen Ngoc	Vi	1/1/2003	vinn@st.usth.edu.vn	1
BI12-019	Doan Bao	Tran	12/3/2002	trandb@st.usth.edu.vn	1

Figure 13. Selection example.

Key points about selection in the relational model:

- Row Filtering: Selection allows you to filter rows from a relation based on specified conditions, enabling the retrieval of relevant and specific information.
- Logical Expressions: Conditions in selection are expressed as logical expressions involving attributes of the relation. Common operators include $=,<,>,<=,>=,$ and $!=$.
- Combining Conditions: Multiple conditions can be combined using logical operators such as `AND, OR`, and `NOT,` allowing for more complex filtering criteria.
- Resulting Relation: The result of a selection operation is a new relation that includes only the rows that satisfy the specified condition. The schema of the resulting relation is the same as the original relation.

d. *Renaming*

Renaming is an operation in the relational model that allows you to assign new names or aliases to relations (tables) or attributes (columns). The purpose of renaming is to provide a more meaningful or concise name for a relation or attribute or to resolve naming conflicts that may arise in the context of a query.

In relational algebra, renaming is denoted by the Greek letter ρ. The general syntax for renaming is:

- To rename a relation from R to S and keep the attributes as they are in R, we write $S = \rho(R)$
- To rename attributes of a relation R we write: $\rho_{\mathrm{S}(\mathrm{A}_1,\mathrm{A}_2,\ldots,\mathrm{A}_n)}(R)$. For example, we can clarify the S relation with the renaming operation: $\rho_{\mathrm{Customer}(ID,Name)}(S)$.

Key points about renaming in the relational model:

- Clarity and Readability: Renaming can improve the clarity and readability of queries by providing more meaningful names for relations and attributes.
- Avoiding Ambiguity: Renaming is useful in situations where multiple relations are involved in a query, and attribute names may be ambiguous. It helps resolve naming conflicts.
- Schema Preservation: Renaming does not alter the schema or the underlying data. It only affects the way relations and attributes are referenced in the context of a specific query.
- Scope of Renaming: The effect of renaming is limited to the scope of the query in which it is used. It does not permanently change the names of relations or attributes in the database schema.

e. Products

In the relational model, the Cartesian Product is an operation that combines every tuple from one relation with every tuple from another relation, resulting in a new relation. The Cartesian Product of two relations, R and S, is denoted as $R \times S$. The resulting relation will have a number of tuples equal to the product of the number of tuples in R and S. Each tuple in the result is formed by combining a tuple from R with every tuple from S.

Key points about the Cartesian Product in the relational model:

- Size of Result: The size of the resulting relation is the product of the number of tuples in the input relations. If R has m tuples and S has n tuples, then $R \times S$ will have $m \times n$.
- No Common Attributes: The Cartesian Product does not require any common attributes between the input relations. It simply combines all possible pairs of tuples.
- Use with Caution: While the Cartesian Product is a legitimate operation, it can lead to a large result set, especially when combining tables with a significant number of tuples. As such, it should be used judiciously.

f. Natural join

A Natural Join is a specific type of join operation in the relational model that combines two relations based on common attributes with the same name. It automatically identifies and joins the tables on columns that share identical names, eliminating the need to specify join conditions explicitly. The result of a natural join includes all columns from both tables, excluding duplicate columns.

In mathematical terms, if R and S are two relations, the natural join $R \bowtie S$ is performed by matching the columns with the same names in both relations. The result includes all columns from both R and S, with duplicate columns removed.

Key points about natural join in the relational model:

- Automatic Matching: Natural join automatically identifies and matches columns with the same names in both tables. There's no need to specify join conditions explicitly.
- Resulting Columns: The result of a natural join includes all columns from both tables, with duplicate columns removed. The common attribute used for the join is included only once in the result.
- Implicit Equality Condition: The natural join implicitly uses equality conditions on columns with the same names, creating a more concise syntax.
- Limited Flexibility: While natural joins offer simplicity, they lack the flexibility of explicitly specifying join conditions. In cases where columns with the same name are not intended for joining, the natural join may produce unexpected results.
- Ambiguity Handling: Ambiguities may arise if the tables have columns with the same name that are not intended for joining. Some database systems provide mechanisms to handle such ambiguities, such as using aliases or disambiguating column names.

g. *Theta join*

A theta join is a type of relational join operation in the relational model that allows for more general join conditions by using any comparison operator to define the relationship between two tables. In mathematical terms, if R and S are two relations, and θ is a condition involving attributes from both relations, then the result of the theta join $R \bowtie_{\theta} S$ is a new relation containing all tuples that satisfy the specified condition θ. This operation is constructed as follows:

1. Compute the product of R and S.

2. Choose the tuples from the product that meet the condition θ.

Theta joins are powerful because they offer more flexibility in defining relationships between tables, allowing for a broader range of comparisons beyond simple equality. However, they also require careful consideration to ensure that the chosen conditions make sense in the context of the data being queried.

6. Two sides of the relational model

a. Advantages

The relational data model offers several advantages, making it a widely adopted and foundational approach in database management. Here are the main advantages of the relational data model:

- Simplicity and Intuitiveness: the tabular structure of the relational model, with rows and columns, is easy to understand and work with. This simplicity facilitates data modeling, database design, and query formulation.
- Data Integrity and Accuracy: the use of primary keys ensures the uniqueness of each row in a table, preventing duplication and ensuring data accuracy. Foreign keys establish relationships between tables, enforcing referential integrity and maintaining consistency across the database.
- Flexibility and Adaptability: the relational model is flexible and can adapt to changing data requirements. New tables can be added, existing tables modified, and relationships adjusted without significant impact on the overall database structure.
- Efficient Querying with SQL: the Structured Query Language (SQL) provides a standardized and powerful interface for interacting with relational databases. SQL enables users to retrieve, update, and manipulate data in a declarative and efficient manner.
- Scalability and Performance: Relational databases can scale to handle large datasets and are optimized for efficient data retrieval and manipulation. Indexing and query optimization techniques contribute to performance enhancements.

These advantages collectively make the relational data model a robust and widely adopted solution for organizing, managing, and querying structured data in various domains and industries. Although it was proposed many years ago, it is still the most common data model used in databases nowadays. In this chapter, we will examine the fundamental concepts of the relational data model.

b. Disadvantages

While the **relational model** has many advantages, it also has some limitations that can affect its suitability for certain applications

- Schema Rigidity: the relational model requires a predefined schema, which can make it inflexible in environments where data structures frequently change. Altering the schema can be complex and time-consuming, impacting database performance and availability during changes.

- Complexity in Design: Designing a relational schema can be intricate as defining relationships, keys, and normalization requires careful planning. Moreover, modeling many-to-many relationships and other complex associations can be cumbersome, often requiring additional tables (junction tables) and more complex queries, which can affect both performance and readability.
- Scalability Issues: Relational databases can struggle with horizontal scalability, which is the ability to add more servers to handle increasing loads. As data volume grows, the performance of read and write operations can degrade, making it challenging to maintain efficiency in large-scale distributed environments.
- Limited Flexibility in Data Types: Some relational databases have limitations on the types of data they can store and process efficiently. Advanced data types, such as spatial, temporal, and JSON, may have limited support or require additional configurations and extensions.

These limitations highlight why alternative database models, such as NoSQL databases, and others, have emerged to address specific needs that relational databases may not handle as effectively.

Highlight & Summary

The chapter provides a comprehensive overview of the relational data model, from its historical roots to its practical applications in modern database management. By understanding the principles and components of the relational model, database practitioners gain insights into designing, implementing, and querying relational databases effectively.

Exercises

1. Study the relation “AcademicYear” in section 4.a, describe the following terms of this relation: attributes, schemas, tuples, domains, relation instance, key of relation?

2. Explain the purpose of using a projection in a relational database query.

3. Describe the syntax for performing a selection operation in SQL.

4. Explain how the selection operation helps in filtering data from a database table.

5. What are some common scenarios where renaming is particularly useful?

6. Explain the purpose of using a product operation in a relational database query.

7. What are some common scenarios where the natural join operation is used in database queries?

8. How does a theta join differ from natural join?

Chapter 3: Structured Query Language

In the realm of modern data management, Structured Query Language (SQL) serves as the standard language to work with DBMSs. Whether you are a seasoned database administrator, a software developer, a data analyst, or just embarking on your journey in the world of data, SQL is a language you simply cannot ignore.

This chapter serves as a gateway to the exciting and powerful world of SQL, where we will explore the fundamental concepts, syntax, and capabilities of this universally adopted language. A quick reference of SQL can be found at [4].

By the end of this chapter, the learners should have a solid understanding of SQL, specially achieve the following goals:

- Understand the concept and philosophy of SQL.
- Know milestones in the development of SQL.
- Able to recognize the type of SQL statements.
- Able to write DDL statements to define objects in a relational database.
- Able to write DML statements to manipulate data in a relational database.

1. SQL Overview

Chapter 2 introduces relational algebra (RA) which defines some mathematical operations to extract data from existing relations. However, computer practitioners may find it inconvenient to write these mathematical notations. Moreover, relational algebra is insufficient to work with real databases as lacks of operations for modifying the relations. As a result, the real DBMSs need a programming language to let users manipulate data. While there exist many programming languages out there for building applications, we are lucky to know that SQL is the standardized domain-specific programming language that serves as the lingua franca for interacting with relational database management systems (RDBMS). Figure 14 compares the way we manipulate databases by RA or SQL. By the end of this chapter, we will see that while RA only defines the way we query data from relations; SQL is more powerful as it supports us in performing the full CRUD operators (Create, Read, Update, Delete) on an existing table, define or change database structure with different objects (tables, indices, views, triggers…), and even write logical and procedural blocks like some general-purpose programming languages such as Java, C…

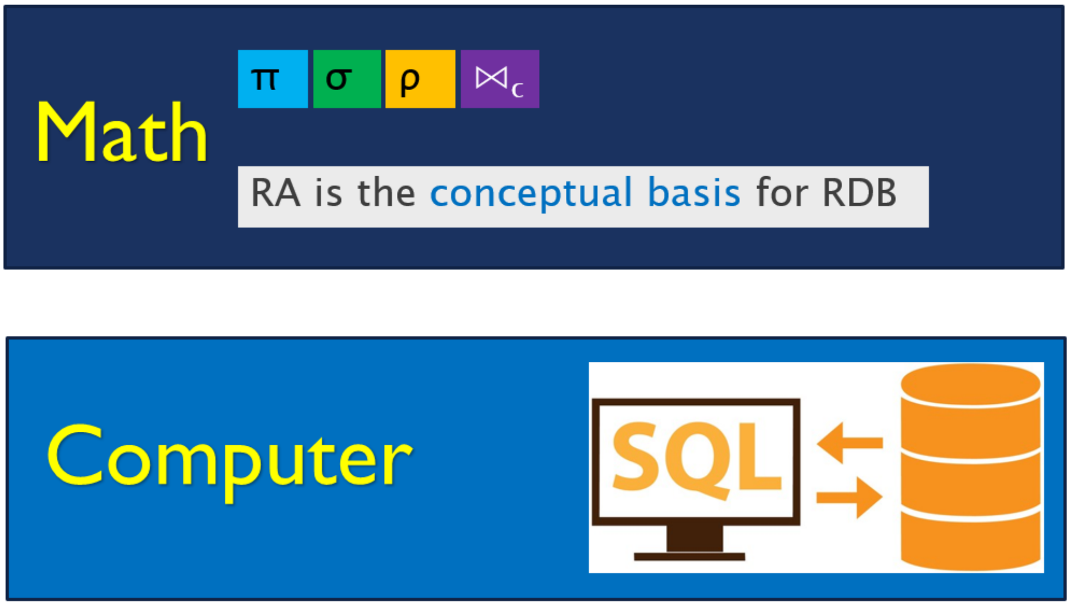


Figure 14. The ways to manipulate data from different points of view: in Math and in practice with computer.

In fact, SQL is not a complete programming language but rather a sublanguage dedicated to working with data. While many programming languages require programmers to declare variables and data structures and implement algorithms, SQL works differently in the sense that SQL is a declarative language. That means users specify what they want to retrieve or manipulate, not how to achieve it. This philosophy aligns with the idea that database management should abstract the complexities of data storage and retrieval, allowing users to focus on the information they need.

Another core philosophy of SQL is standardization. SQL has established ANSI/ISO standards to ensure portability across different database management systems. This philosophy is vital for data consistency and interoperability in a world where various databases coexist. Although SQL is a standard and common language for all RDBMSs, they have their own variations or dialects of SQL, often adding proprietary features and extensions to the standard SQL to enhance functionality and performance.

With SQL, users can request a DBMS to perform data-related tasks by submitting SQL statements to the database server and then waiting for the response. In this book, we use MySQL as the environment for demonstrating SQL (see section 3 for how to work with MySQL). If readers would like to perform these examples in another DBMS, carefully looking up the specific SQL dialect in the official documentation is suggested.

2. The Evolution of SQL

Structured Query Language, or SQL, has come a long way since its inception in the 1970s. Over the decades, it has evolved to meet the ever-expanding demands of data management, making it one of the most widely used and enduring programming languages in the world. Let us take a journey through the key milestones in the evolution of SQL.

SQL's roots can be traced back to the IBM Research Laboratory in San Jose, California, where the language was born in the early 1970s. The lab aimed to provide a more user-friendly and structured way to query and manipulate data in databases. Initially, it was named SEQUEL (Structured English Query Language) so nowadays, SQL is sometimes still pronounced "sequel" as a habit. SEQUEL's precursor was used within IBM's System R project, which laid the foundation for modern relational database management systems (RDBMS).

As SQL usage grew beyond IBM's System R, various database vendors began implementing their own versions of SQL. To address the need for a common standard, the American National Standards Institute (ANSI) published the first SQL standard in 1986. This standardization marked a crucial turning point, ensuring that SQL was no longer limited to a single vendor or platform but could be adopted universally.

Subsequent revisions of the SQL standard introduced new features and enhancements. SQL-92, released in, as the name suggests, 1992, brought significant improvements to the language, including support for more complex queries and data manipulation operations. Subsequent iterations, such as SQL:1999, SQL:2003, and SQL:2008, continued to expand SQL's capabilities, introducing features like recursive queries, user-defined types, and window functions.

While SQL remains at the core of many data management systems, the rise of NoSQL databases in the early 21st century posed new challenges and opportunities. NoSQL databases, with their flexible data models, questioned the dominance of traditional relational databases. In response, SQL evolved to accommodate these changes, with support for JSON data, spatial data, and other non-relational data types.

Open source databases like MySQL, PostgreSQL, and SQLite have played a pivotal role in the widespread adoption of SQL. These databases not only made SQL accessible to a broader audience but also contributed to the development of new SQL features and extensions. The SQL community, including many developers and database administrators, has actively shaped the language's evolution by contributing to open source projects and standards committees.

As organizations grapple with massive volumes of data, SQL has adapted to meet the needs of big data and analytic. Analytical SQL extensions and specialized platforms have emerged, enabling complex data analysis and processing. SQL-on-Hadoop solutions, like Apache Hive and Apache Impala, have further expanded the horizons of SQL, allowing it to interact with vast datasets stored in distributed file systems.

SQL continues to evolve, embracing modern developments such as cloud databases, machine learning integration, and advanced security features. The versatility and adaptability of SQL make it a key player in the ongoing data revolution, ensuring that it remains relevant in a constantly changing data landscape.

This evolution of SQL reflects not only its resilience but also its ability to meet the evolving needs of data management and analytics. As we progress through this chapter and the subsequent chapters in this book, you'll gain a deeper understanding of the latest capabilities and best practices in SQL, equipping you to excel in the dynamic field of data management.

3. SQL Environment

This section describes how SQL statements work with other software components of an information system. Shown in Figure 15 is a typical setup of an information system which consists of applications at the client side, and a DMBS along with its databases at the server side.

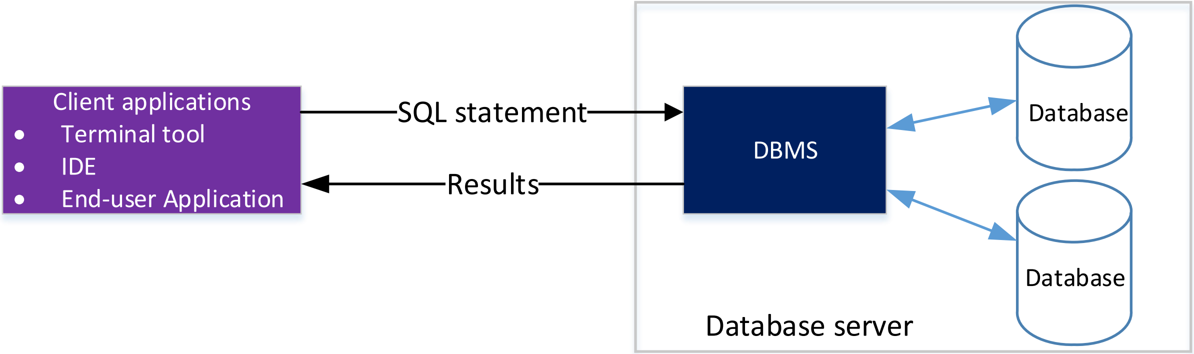


Figure 15. SQL environment.

The client applications are tools or programs used by different database stakeholders (software developers, DB administrators, IT support, end-user) to interact with databases via DBMS. They allow user to edit or type SQL statements, send them to the DMBS, then receive and display the results.

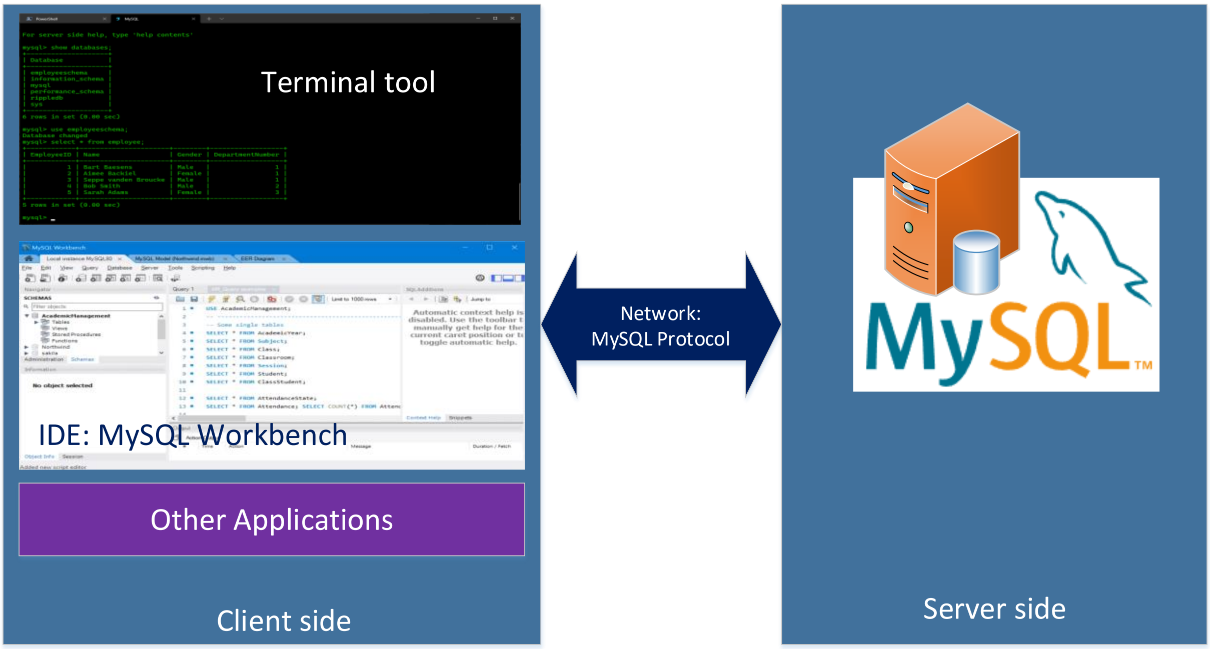
There are three types of client applications. The first one is Terminal tool a.k.a Command-line interface (CLI), which often comes with database software bundles. The Integrated Development Environment (IDE) tools are mainly used by software development teams to develop and test SQL statements. Examples of DB IDE include MySQl Workbench, phpMyAdmin, pgAdmin, Microsoft SQL Server Management Studio, Visual Studio. The third application type is end-user applications which are tailored to fit business functionalities; therefore, they often hide all the SQL commands to the users. When an action such as opening a form, or clicking the save button is performed, the end-user applications issue SQL commands automatically and show the query result in its own format specialized for businesses. The End-user Applications are prewritten in either web-form, mobile application, or PC application form.

4. Getting started with MySQL

a. Introduction

MySQL is a widely-used open-source relational database management system (RDBMS) that is known for its reliability, speed, and ease of use. Developed by Oracle Corporation, MySQL supports a broad range of applications, from small, single-user projects to large-scale, enterprise-level applications. Individuals and enterprises can freely use, modify, publish, and extend Oracle's open-source MySQL codebase. The software is distributed under the GNU General Public License (GPL). It is widely compatible with all major platforms, including Windows, Linux, and macOS. MySQL is highly valued for its robust performance, scalability, and extensive support for various storage engines, which allow users to customize database performance and storage options. Its popularity is bolstered by a strong community and comprehensive documentation, making it an excellent choice for both beginners and experienced developers seeking to implement efficient and effective database solutions.

b. Working environment



```
For server side help, type 'help contents'

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| employeeschema     |
| information_schema |
| mysql              |
| performance_schema |
| rippledb           |
| sys                |
+--------------------+
6 rows in set (0.00 sec)

mysql> use employeeschema;
Database changed
mysql> select * from employee;
+------------+----------------------+--------+------------------+
| EmployeeID | Name                 | Gender | DepartmentNumber |
+------------+----------------------+--------+------------------+
|          1 | Bart Baesens         | Male   |                1 |
|          2 | Aimee Backiel        | Female |                1 |
|          3 | Seppe vanden Broucke | Male   |                1 |
|          4 | Bob Smith            | Male   |                2 |
|          5 | Sarah Adams          | Female |                3 |
+------------+----------------------+--------+------------------+
5 rows in set (0.00 sec)

mysql> _
```

```
USE AcademicManagement;
-- ------------------------------------------------
-- Some single tables
SELECT * FROM AcademicYear;
SELECT * FROM Subject;
SELECT * FROM Class;
SELECT * FROM Classroom;
SELECT * FROM Session;
SELECT * FROM Student;
SELECT * FROM ClassStudent;

SELECT * FROM AttendanceState;
SELECT * FROM Attendance; SELECT COUNT(*) FROM Attenc
```

Automatic context help is disabled. Use the toolbar t manually get help for the current caret position or t toggle automatic help.

No object selected

Figure 16. My SQL working components.

Like other RDBMSs, MySQL operates within a client-server architecture, where the MySQL server is a service or daemon process (see Figure 16). It runs continuously in the background, handling all database operations, including query processing, data management, and user authentication. It listens for client requests on a specific network port (the default is port 3306) and communicates over the network using the MySQL protocol.

On the client slide, there are tools and interfaces that allow users to interact with the MySQL database server. These applications enable users to execute SQL queries, manage databases, and perform various administrative tasks. The MySQL command-line client (mysql) is a standard tool that comes with MySQL distributions. It allows users to execute SQL queries and perform database operations through a command-line interface. The preferable tool for developers is MySQL Workbench, while there are alternatives such as phpMyAdmin and Visual Studio Code. Applications written in various programming languages (such as PHP, Java, Python) use MySQL connectors and APIs to interact with the MySQL server. In a typical web application, the web server (acting as a client) runs a PHP script that needs to retrieve data from the database. The PHP script sends an SQL query to the MySQL server. The MySQL server processes the query, fetches the requested data, and sends it back to the PHP script, which then formats and displays the data to the user through a web browser.

c. *Setting up the server*

Installing MySQL on Windows

- Step 1: Download MySQL installer at the Official Website: https://dev.mysql.com/downloads/installer/
- Step 2: Run the Installer, follow the on-screen instructions provided by the setup wizard, choose the setup type. For most users, the "Developer Default" setup is recommended. The installer will download and install the required components. This process may take a few minutes.
- Step 3: Configure MySQL: After the installation is complete, choose the configuration type (e.g., "Development Computer"), set the root password and, optionally, create additional MySQL user accounts, then configure the MySQL server as a Windows service.

Installing MySQL on macOS

- Step 1: Download MySQL DMG Package at the Official Website: https://dev.mysql.com/downloads/installer/
- Step 2: Install MySQL: follow the on-screen instructions to install MySQL (the macOS administrator password may be asked).
- Step 3: Configure MySQL: after the installation is complete, you will be prompted to initialize the database. You then need to set the root password and choose any additional configuration options. MySQL will be configured to start automatically. You can manage the MySQL server from the "System Preferences" pane under "MySQL".

Installing MySQL on Linux

- Step 1: Install MySQL

```
sudo apt-get install mysql-server
```

- Step 2: Start MySQL Service:

```
sudo systemctl start mysql
```

- Step 3: Log in to MySQL with the sudo command.

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH
mysql_native_password BY '[password]';
```

- Step 4: Change the password of root accoun:

d. Verifying Installation

After installing MySQL, you should verify that the installation was successful.

- Open the command line in Windows or Terminal in MacOS/Linux, then connect to MySQL by command:

```
mysql -u root -p
```

- Check MySQL Version:

```
SELECT VERSION();
```

You can run some commands to examine data at the server.

- Listing the database:

```
show databases;
```

- Change the default database:

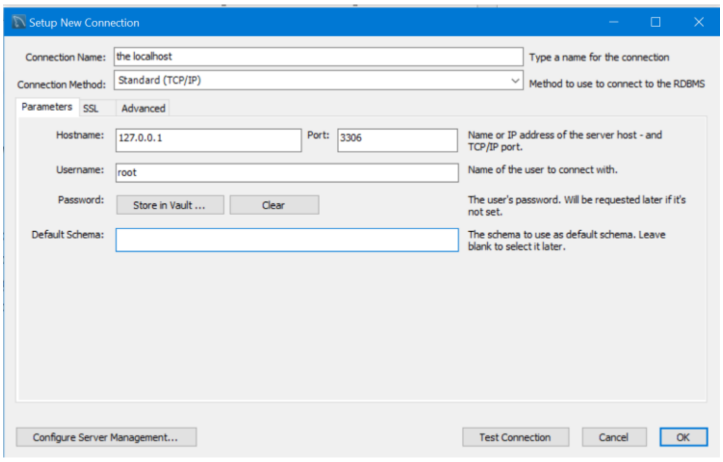
```
use <database_name>;
```

- List tables of the current database:

```
Show tables;
```

e. Using MySQL Workbench

Creating a connection cofiguration



Connection Name: the localhost — Type a name for the connection

Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Hostname: 127.0.0.1 Port: 3306 — Name or IP address of the server host - and TCP/IP port.

Username: root — Name of the user to connect with.

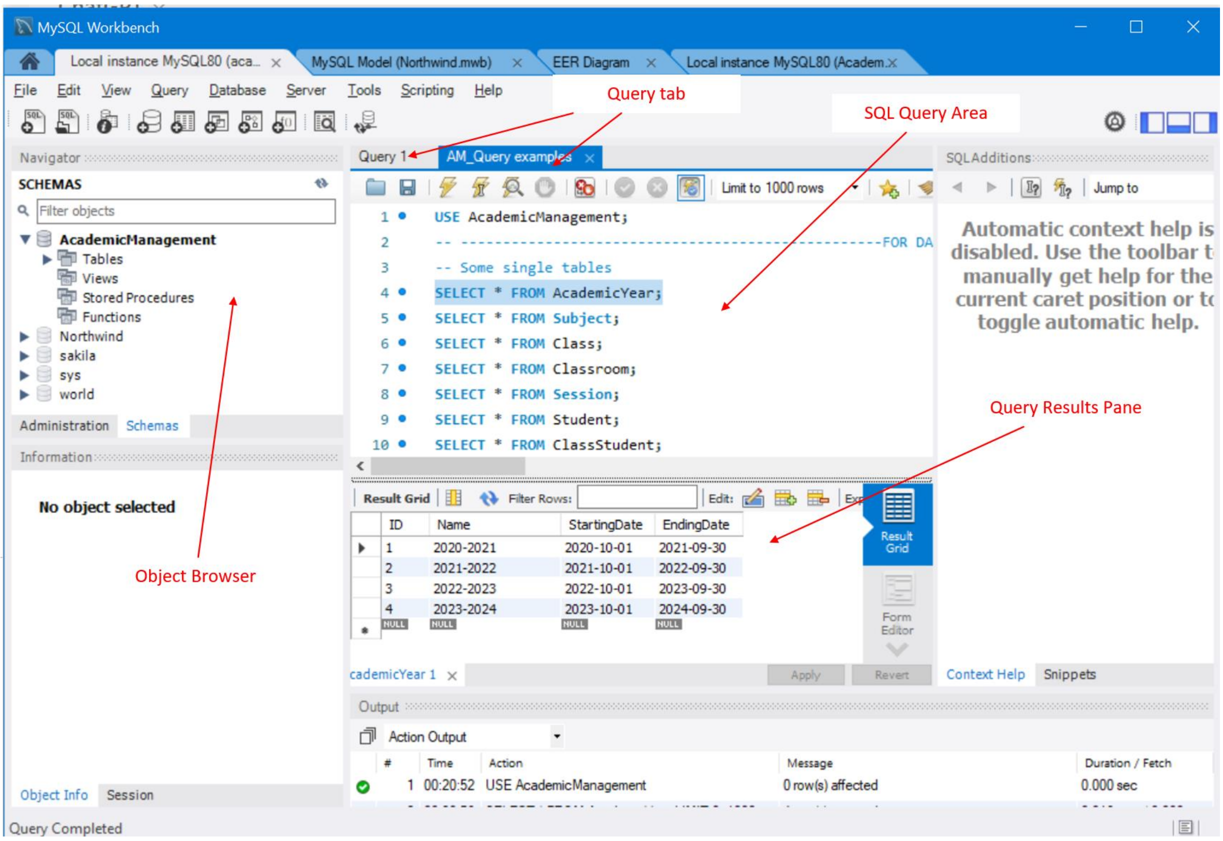
Password: — The user's password. Will be requested later if it's not set.

Default Schema: — The schema to use as default schema. Leave blank to select it later.

Figure 17. New Connection screen.

At the welcome screen of MySQL Workbench, click on the plus sign (+) to open the screen defining the connection. User need to define connection name, hostname which is the server address on the network, the port, and username (see Figure 17). The defined configuration can be used at the next time to connect to the desired server.

Developing SQL statements



```
USE AcademicManagement;
-- ---------------------------------------------------------FOR DA
-- Some single tables
SELECT * FROM AcademicYear;
SELECT * FROM Subject;
SELECT * FROM Class;
SELECT * FROM Classroom;
SELECT * FROM Session;
SELECT * FROM Student;
SELECT * FROM ClassStudent;
```

Automatic context help is disabled. Use the toolbar t manually get help for the current caret position or tc toggle automatic help.

ID	Name	StartingDate	EndingDate
1	2020-2021	2020-10-01	2021-09-30
2	2021-2022	2021-10-01	2022-09-30
3	2022-2023	2022-10-01	2023-09-30
4	2023-2024	2023-10-01	2024-09-30
NULL	NULL	NULL	NULL

#	Time	Action	Message	Duration / Fetch
1	00:20:52	USE AcademicManagement	0 row(s) affected	0.000 sec

Figure 18. The GUI of MySQL Workbench for SQL statement development.

The graphical user interface (GUI) of MySQL Workbench is designed to develop, manage, and execute SQL statements through a user-friendly environment (Figure 18). Some notable areas on the GUI are:

- Object Browser is the left panel which helps user to explore and interact with database objects.
- The Query Tabs allows multiple SQL scripts to be open simultaneously in separate tabs. Each tab can contain a different SQL script or query, making it easy to switch between different tasks or queries.
- SQL Query Area is main workspace for writing SQL statements. It supports syntax highlighting, which colors different parts of the SQL syntax to improve readability and reduce errors. It suggests SQL keywords, table names, column names, and functions as users type, helping to speed up development and reduce errors.
- Query Results Panel displays the results of executed SQL queries and provides feedback on query performance.

5. Coding with SQL

a. Elements of SQL code

An SQL **statement** is a complete instruction to the database to perform a specific task. For example, the statement

```
SELECT * FROM Employees
WHERE LastName = 'Smith'
ORDER BY FirstName;
```

can be used to query data from table Employees in the current database. Each statement normally ended by a semicolon “;”. The taxonomy of statement is fully explained at section 5.

A **clause**, on the other hand, is a part of an SQL statement. Clauses are used to specify conditions or to provide more detail to the statement. They refine and define the operations performed by the SQL statement. Clauses often work together within a statement to provide specific functionality. The above discussed statement consists of 3 clauses namely “`SELECT * FROM Employees`“, “`WHERE LastName = 'Smith'`“, AND “`ORDER BY FirstName`”; each normally is written in a line for the sake of readability.

In SQL, **keywords** are reserved words that have a predefined meaning in the language. These keywords are used to perform various operations on databases, such as querying data, defining structures, controlling access, and managing transactions. SQL keywords are fundamental to constructing SQL statements and clauses.

Identifiers in SQL are names used to identify various database objects such as tables, columns, indexes, views, schemas, and other elements within a database. These names must follow specific rules and conventions to ensure they are recognized correctly by the database management system (DBMS). There are different types of identifiers:

- Table Names: The names given to tables.
- Column Names: The names given to columns within tables.
- Index Names: The names given to indexes.
- View Names: The names given to views.
- Schema Names: The names given to schemas.

When an identifier has no space, reserved keywords and special characters, we can write them without quotes. Readers can see the previous statement at the beginning section to verify this rule since `Employees`, `LastName`, and `FirstName` are all identifiers with only normal characters. In contrast, it must be enclosed with double quote (or square brackets in SQL Server). For example, the quotes must be written in the following statement:

```
SELECT * FROM "Employees IN Company"
WHERE "Last Name" = 'Smith'
ORDER BY "First Name";
```

b. Case Insensitivity

SQL is SQL is generally case-insensitive, meaning that it does not distinguish between uppercase and lowercase letters in keywords, table names, column names, and other identifiers. However, there are nuances and variations depending on the specific SQL implementation and configuration. Here are some key points to consider:

- SQL keywords such as `SELECT`, `FROM`, `WHERE`, `INSERT`, etc., are case-insensitive. This means you can write them in uppercase, lowercase, or a mixture of both without affecting the functionality. For example, `SELECT * FROM table_name` is equivalent to `select * from table_name`.
- Table names, column names, and other identifiers are generally case-insensitive, but this can vary by database system and settings.
- In some systems, such as MySQL, the case sensitivity of table names and column names depends on the underlying operating system. For instance, Windows file systems are case-insensitive, so MySQL on Windows is case-insensitive by default. On the other hand, Linux file systems are case-sensitive, making MySQL case-sensitive on Linux unless configured otherwise.

c. Coding convention

Coding conventions in SQL help maintain readability, consistency, and maintainability of code across different projects and teams. Here are some widely accepted conventions:

Use Consistent Case

- Keywords: uppercase (`SELECT, FROM, WHERE, …`).

- Identifiers (tables, columns): lowercase with underscore to separate words (`table_name`, `column_name`), or PascalCase (TableName, ColumnName).

Naming convention

- Tables: Singular nouns (customer, order).
- Columns: Descriptive names (first_name, order_date).

Indentation and Alignment

- Align SQL statements and clauses for better readability. Each clause normally is written in a line.
- Use indentation to indicate nested or queries

```
SELECT column_name
```

```
FROM (
  SELECT column_name
  FROM table_name
  WHERE condition
) AS subquery;
```

Commenting

Use **inline comments** for complex logic or calculations.

```
SELECT column_name -- This is a comment
FROM table_name;
```

Use **block comments** for detailed explanations.

```
/*
  This is a block comment.
  It can span multiple lines.
*/
SELECT column_name
FROM table_name;
```

6. Taxonomy of SQL statements

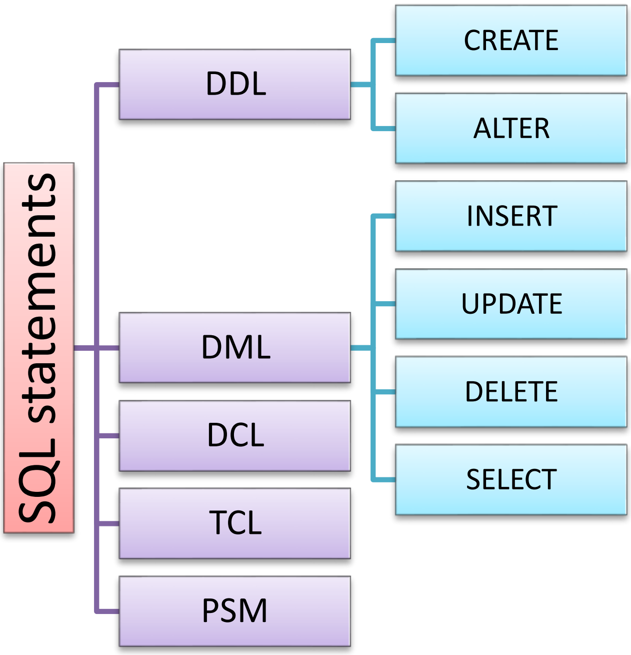


Figure 19. The SQL statement taxonomy.

SQL statements can be classified into different categories based on their functionalities and purposes [5] as illustrated in Figure 19. First, the Data Definition Language (DDL) commands are used to create databases, tables and their relationship, and other objects inside databases. Second, the Data Manipulation Language (DML) statements deal with manipulating (inserting, updating, deleting, querying) rows of tables inside databases. Next, the Data Control Language (DCL) commands are used to control security and access to data, grant or revoke permissions to users or groups. Transaction Control Language (TCL) is used to manage transactions in databases. Finally, Persistent Stored Module (PSM) commands is an extension of the SQL to enable procedural programming feature like control-flow statements, using variable and cursor.

The upcoming sections will explain these categories in details.

7. DDL statements

DDL, which stands for Data Definition Language, encompasses SQL commands specifically designed for defining a database's schema. It primarily involves working with schema descriptions and is employed to establish and alter the structure of database elements within the database. Specially, DDL commands are used to define, change, and delete all object types in a DBMS including databases, tables, views, indexes, stored procedure, user-defined functions, trigger,...

These instructions are typically not utilized by ordinary users, who should access the database through applications but the professional database users such as database admins, programmers.

a. Defining objects with CREATE statement

All the objects in a DBMS are defined with the CREATE command. Its general syntax is:

```
CREATE object_type object_name;
```

where:

```
<object type> = DATABASE | TABLE | INDEX | VIEW | PROCEDURE | TRIGGER
```

Before creating any objects (table, view, procedure) you must create a database by following statement.

```
CREATE DATABASE [IF NOT EXISTS] database_name [create_option];
```

For example, to create a database titled AcademicManagement, you can simply run the following statement:

```
CREATE DATABASE AcademicManagement;
```

The clause `IF NOT EXISTS` is useful to make sure the database does not exist in advance. Once the database exists, users can explicitly change the database to work with by the statement:

```
USE database_name;
```

For example: `USE AcademicManagement;` will set the current default database to "AcademicManagement".

Then, users can define tables for the current database by the `CREATE TABLE` statement with following syntax:

```
CREATE TABLE table_name(

    column_name_1 datatype [column_level_constraints],

    column_name_2 datatype [column_level_constraints],

    ...

    [constraints]

);
```

Now, let us define tables Department and Lecturer. While the Primary Key of each table is clear, each table has a foreign key referring to the primary of the other. More detail, each teacher may be long to a department in case he/she is a permanent lecturer so the Teacher.DepartmentID is a foreign key. On the other hand, each department may has 0 or 1 director who is actually on the list of Teacher so the Department.DirectorID is a foreign key. Since two tables depend recursively, we cannot define all the relationships between tables at the time we create them. If we create the table Department first, there is no table named Teacher to refer to, so we must add this referential constraint to Department once the Teacher table exists by using `ALTER TABLE ADD CONSTRAINT` (read more at the next section).

Now the table Department can be defined without the foreign key as below:

```
CREATE TABLE IF NOT EXISTS Department (

    ID INT NOT NULL PRIMARY KEY,

    Name VARCHAR(50) DEFAULT NULL,

    MainOffice varchar(30) DEFAULT NULL,

    DirectorID INT NULL,

    PRIMARY KEY (`ID`)
);
```

Table Teacher, however, can be created with it full constraints:

```
CREATE TABLE IF NOT EXISTS Teacher (

  ID int NOT NULL,

  FirstName varchar(45) NOT NULL,

  LastName varchar(45) NOT NULL,

  SSN char(12) DEFAULT NULL,

  Birthdate date DEFAULT NULL,

  Email varchar(50) DEFAULT NULL,

  Rank char(10) DEFAULT NULL,

  Affiliation varchar(50) DEFAULT NULL,

  DepartmentID int DEFAULT NULL,

  PRIMARY KEY (ID),

  FOREIGN KEY (DepartmentID) REFERENCES Department(ID)

                             ON DELETE SET NULL ON UPDATE CASCADE

);
```

This section does not dig into the details of how to define other object types like index, view, procedure, and trigger; however, ones can check manual of each DBMS for the details.

b. Other DDL statements

Once objects such as databases and tables are created, people sometimes need to change or remove them. The below statements do the jobs:

- `ALTER object_type object_name:` This is used to modify the existing objects, such as adding columns to a table. Specially with ALTER TABLE statement, some actions are available:
  - `ADD [COLUMN] column_definition`
  - `ALTER [COLUMN] column_name`
  - `DROP column_name`
  - `ADD table_constraint`

- `DROP object_type object_name`: This command is used to remove the existing object.
- `RENAME`: This is used to rename an object
- `TRUNCATE [TABLE] table_name`: This is used to remove all records from a table, including all spaces allocated for the records are removed. In fact, this statement first removes the table and then creates a new one with the same structures as the original. That explains why `TRUNCATE` is classified into the DDL statements.

Let us recall that in the previous section, the Department table has been defined but the foreign key from the column DirectorID to Teacher.ID is missing. We now add this constraint by the following statement:

```
ALTER TABLE Department

ADD FOREIGN KEY (DirectorID) REFERENCES Department(ID)

                             ON DELETE SET NULL ON UPDATE SET NULL;
```

Sometimes, we learn that a column should be supplemented. For example, the phone number can make teacher data richer, we can run the below statement:

```
ALTER TABLE Teacher

ADD COLUMN Phone VARCHAR(15);
```

8. DML statements

a. Adding, modifying, and removing rows

Once a table has been created, people use it to hold records. Inserting (a) record(s) can be done by the INSERT statements with the following syntax:

```
INSERT INTO table_name (column_1, column_2, ...)

VALUES  (value_1_1, value_1_2, ...),

        (value_2_1, value_2_2, ...);
```

In that statement, the number, as well as the data types of columns and values must match for each row inserted. People can insert one record or perform a batch insert with multiple rows with one `INSERT` statement. For example, the two rows of table student can be populated by the following statement:

```
INSERT INTO Department (ID, Name, MainOffice)

VALUES (1,'General Education Department','A21 201'),
        (2,'Foreign Language Center','A21 707'),

        (3,'Information and Communication Technology Department','A21 402');
```

The rows of a table can be removed by the DELETE statement with the general syntax:

```
DELETE FROM table_name

[WHERE condition];
```

The WHERE condition is optional. If no WHERE condition is specified, all the rows will be removed, which is similar to the TRUNCATE statement discussed earlier. The condition is a Boolean expression that can be True or False with specific values of each record. All the records resulting in the True value will be removed.

To update the data of records, the statement UPDATE needs to specify which records and the new values of columns. Its syntax is as below:

```
UPDATE table_name

SET column_1 = value_2, column_2 = value_2 ...

[WHERE condition];
```

For example, if we would like to use a short name in abbreviation instead of the long name of one department, the following statement is the solution:

```
UPDATE Department

SET Name = 'ICT'

WHERE ID = 3;
```

b. Querying data

After storing data in tables, there exist situations in which we need to retrieve data to view them from different points of view. The most common demand could be preparing data to show on application forms or reports. Programmers also often query data to inspect them. This results in the SELECT statement being the most commonly executed command in SQL. Its general syntax is

```
SELECT [DISTINCT| ALL] select_list,

[FROM table1, table2, ..]

[WHERE conditions,]
```

```
[GROUP BY group_by_expression,]

[HAVING search_condition,]

[ORDER BY column1 [ASC|DESC],  column2 [ASC|DESC],... ];
```

As shown above, this statement comprises 6 clauses, one written in each line for clarification. Among them, only the SELECT clause is mandatory. Although the order of clauses must strictly follow the syntax,

A common good practice is to analyze (when we write or read) these clauses in the following order: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY`.

In the sub-sequence examples, we shell use the database AcademicManagement to demonstrate the uses of the `SELECT` statement.

Basic SELECT ... FROM ... WHERE... ORDER BY query

If the where clause does not appear, all the rows will be returned, so to see the whole Teacher data, run:

```
SELECT * -- The asterisk implies all the columns of tables in the FROM class

FROM Teacher; -- The source of data;
```

The `WHERE` clause is a Boolean expression that can be True or False depending on the value of each row of tables mentioned in the `FROM` clause. For example, to show list of inner lecturers (whose `DepartmentID` is defined clearly), the search_condition should be `DepartmentID IS NOT NULL`. We can specify which columns are returned and their order by defining them in the `SELECT` clause. We can use existing function to manipulate data coming from columns. So the full name (`FirstName` concatenating with `LastName`) and email of inner teachers are:

```
SELECT CONCAT(LastName, ' ', LastName), Email

FROM Teacher

WHERE DepartmentID IS NOT NULL;
```

The below statement will show the list of invited professors:

```
SELECT CONCAT(LastName, ' ', LastName), Email, Affiliation

FROM Teacher

WHERE Rank = 'Prof.' AND DepartmentID IS NULL;
```

The order of returned rows by default depends on the physical data structure storing the table. If this is desirable, the ORDER BY clause can sort the rows by indicating the column and the direction (ASC for ascending and DESC for descending). For example, we can sort the list of teachers by their departmentID and name as follows:

```
SELECT DepartmentID, CONCAT(LastName, ' ', LastName), Email, Affiliation

FROM Teacher

ORDER BY DepartmentID DESC, LastName ASC, FirstName
```

Query multiple tables

In many cases, we need to display data from two or more tables. The resulting rows are formed by matching a row of a table with rows of another table. The cartesian product can be expressed in SQL by list tables in the FROM clause and separating them by comma, for example, Department X Teacher:

```
SELECT Teacher.ID, LastName, FirstName, Name AS DepartmentName

FROM Department LEFT JOIN  Teacher Department.ID = Teacher.ID;
```

In practice, this simple form of query is rarely used; instead, the Inner join is much more common. For example, we would like to show each teacher with his/her department name if matched data is available. As discussed in relational algebra, the inner join is the result of applying selection on a cartesian product, so adding the matching criteria to the WHERE clause is a way to express the INNER JOIN.

```
SELECT *

FROM Department, Teacher

WHERE Department.ID = Teacher.ID;
```

However, many DBMSs support a more explicit way to express the JOIN operation. That is writing the key word JOIN in the FROM clause, for example:

```
SELECT Teacher.ID, LastName, FirstName, Name AS DepartmentName

FROM Department INNER JOIN  Teacher Department.ID = Teacher.ID;
```

Please noted that with INNER JOIN, only matched rows are shown. However; sometimes, our demand is to include rows of the original table even when there is no matched counterpart. For example, beside matched rows, we show department names whose no teacher belong to:

Or we show some teachers whose DepartmentID is not in the list of DepartmentID.

```
SELECT Teacher.ID, LastName, FirstName, Name AS DepartmentName

FROM Department LEFT JOIN  Teacher Department.ID = Teacher.ID;
```

To show unmatched rows of both table sides, some DBMS including Oracle, SQL Server and PostgreSQL provide keyword `FULL OUTER JOIN`. With MySQL, that keywords is not supported; however, programmers can use `LEFT JOIN` with the UNION of `RIGHT JOIN`. Below is the example of a `FULL OUTER JOIN`:

```
SELECT Teacher.ID, LastName, FirstName, Name AS DepartmentName

FROM Department FULL [OUTER] JOIN  Teacher Department.ID = Teacher.ID;
```

Data Aggregation

Aggregation in SQL refers to the process of performing a mathematical operation on a set of values to derive a single result. Common SQL aggregate functions allow you to perform calculations across rows and summarize data. Here are some commonly used aggregate functions:

- `COUNT`: count the number of rows or non-null values in a column
- `SUM`: calculate the sum of values in a column
- `AVG`: compute the average of values in a column
- `MAX`: find the maximum value in a column
- `MIN`: get the minimum value in a column

Some DBMS can provide some additional functions such as `STDEV, VAR`. Following examples demonstrate the work of aggregate functions. Let us count how many lectures:

```
SELECT COUNT(*)

FROM Lecturers;
```

Show the average age of teachers:

```
SELECT AVG(YEAR(NOW()) - YEAR(BirthDate))

FROM Teacher;
```

In SQL, you can gather rows having common values into groups and then apply aggregate functions on each instead of the whole result set returned by the `WHERE` clause.

Consider table Teacher, we can compute the average age of lecturers from different department.

```
SELECT DepartmentID, AVG(YEAR(NOW()) - YEAR(BirthDate)) AS AverageAge

FROM Teacher

WHERE DepartmentID

GROUP BY DepartmentID;
```

We can filter the result of aggregated values by the clause `HAVING` which requires a Boolean expression like the `WHERE` clause; however, the `HAVING` clause runs after computation of aggregate. For example,if you want to show deparmentIDs whose average age of lecturers are below 35, you could specify:

```
SELECT DepartmentID, AVG(YEAR(NOW()) - YEAR(BirthDate)) AS AverageAge

FROM Teacher

WHERE DepartmentID

    •   GROUP BY DepartmentID

HAVING AverageAge <= 35
```

Subquery

In SQL, a subquery, also known as a nested query or inner query, is a query nested within another query. It allows you to perform complex queries by using the results of one query as a condition or source of data for another query. Subqueries can be used in various parts of a SQL statement, such as the `SELECT, FROM, WHERE, HAVING`, or `JOIN` clauses.

Sub-queries can be classified into 3 categories: single-row, multiple-row, correlated.

Single-Row subqueries returns a single value and can be used with single-row operators like `$=, >, <$`, etc. For example, you want to show list of lecturers from the ICT department:

```
SELECT *

FROM Teacher

WHERE DepartmentID = (SELECT ID

                        FROM Department
```

```
                    WHERE Name = 'ICT');
```

Multiple-Row Subquery: Returns multiple rows and can be used with multiple-row operators like IN, ANY, ALL, etc. For example, you want to show list of lecturers from all the departments (that has the keyword "department" in the name, different from other unit like "Foreign Language Center"), you can run:

```
SELECT *
FROM Teacher
WHERE DepartmentID IN (SELECT ID
                    FROM Department
                    WHERE Name LIKE '%department%');
```

Correlated Subqueries are the case in which the reference columns from the outer query within the subquery, enabling comparison between the inner and outer queries. For example, you can list lecturers who are younger than the average in his/her department with below statement:

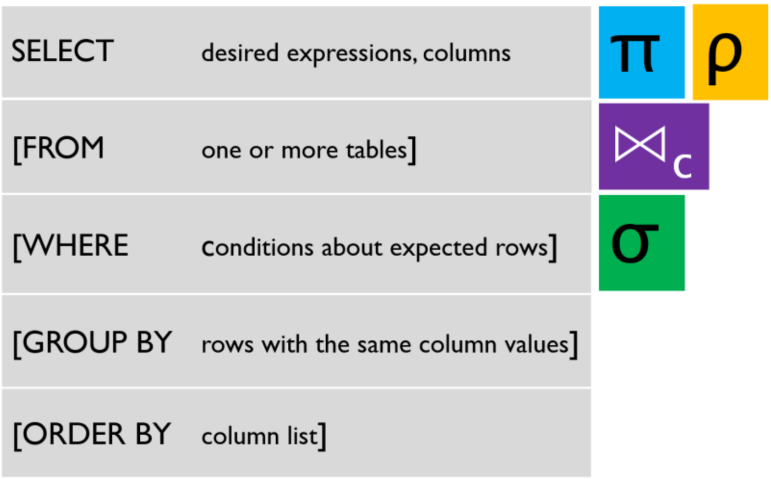
```
SELECT *
FROM Teacher O
WHERE Birthdate  > (SELECT AVG(Birthdate)
                    FROM Teacher I
                    WHERE I.DepartmentID = O.DepartmentID);
```

Subqueries have following uses and advantages:

- Data Manipulation: Subqueries allow for intricate data manipulation by leveraging results from one query to filter, sort, or perform calculations in another.
- Simplifying Complex Queries: They help break down complex problems into smaller, manageable parts, improving query readability and maintenance.
- Optimization and Performance: However, improper usage or nesting of subqueries may impact performance. Using joins or other optimization techniques might be more efficient in certain scenarios.

c. Correspondence of Relational Algebra and SQL

The querying statements of SQL can perform all Relational Algebra operations described in section 5 of chapter 2. Figure 20 points out at which clauses of the SELECT statement we can perform the operations.



SELECT desired expressions, columns

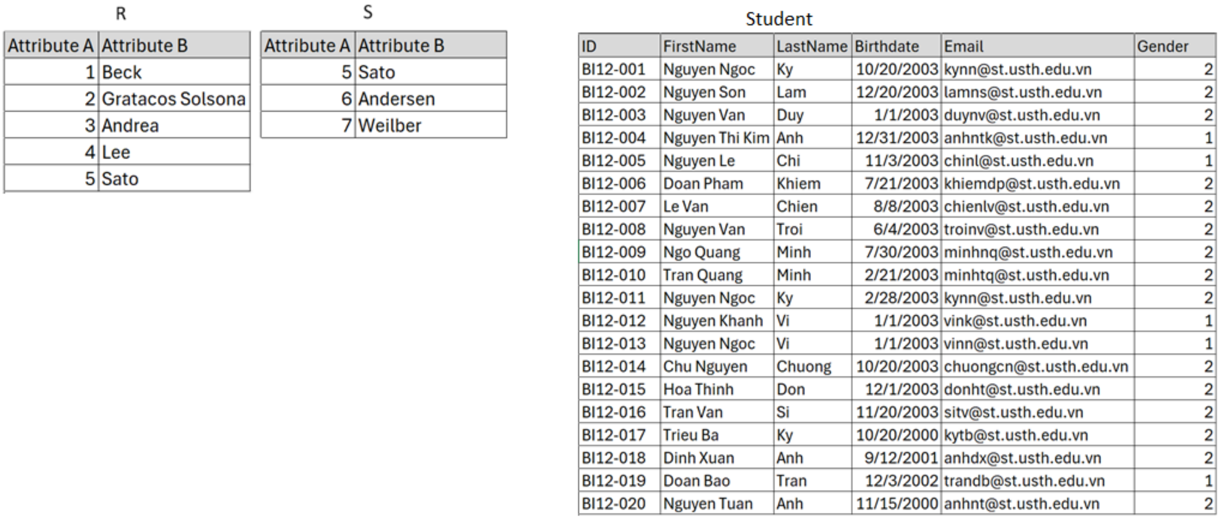
[FROM one or more tables]

[WHERE conditions about expected rows]

[GROUP BY rows with the same column values]

[ORDER BY column list]

Figure 20. The clauses of SELECT statement and corresponding relational algebra operations.



R

Attribute A	Attribute B
1	Beck
2	Gratacos Solsona
3	Andrea
4	Lee
5	Sato

S

Attribute A	Attribute B
5	Sato
6	Andersen
7	Weilber

Student

ID	FirstName	LastName	Birthdate	Email	Gender
BI12-001	Nguyen Ngoc	Ky	10/20/2003	kynn@st.usth.edu.vn	2
BI12-002	Nguyen Son	Lam	12/20/2003	lamns@st.usth.edu.vn	2
BI12-003	Nguyen Van	Duy	1/1/2003	duynv@st.usth.edu.vn	2
BI12-004	Nguyen Thi Kim	Anh	12/31/2003	anhntk@st.usth.edu.vn	1
BI12-005	Nguyen Le	Chi	11/3/2003	chinl@st.usth.edu.vn	1
BI12-006	Doan Pham	Khiem	7/21/2003	khiemdp@st.usth.edu.vn	2
BI12-007	Le Van	Chien	8/8/2003	chienlv@st.usth.edu.vn	2
BI12-008	Nguyen Van	Troi	6/4/2003	troinv@st.usth.edu.vn	2
BI12-009	Ngo Quang	Minh	7/30/2003	minhnq@st.usth.edu.vn	2
BI12-010	Tran Quang	Minh	2/21/2003	minhtq@st.usth.edu.vn	2
BI12-011	Nguyen Ngoc	Ky	2/28/2003	kynn@st.usth.edu.vn	2
BI12-012	Nguyen Khanh	Vi	1/1/2003	vink@st.usth.edu.vn	1
BI12-013	Nguyen Ngoc	Vi	1/1/2003	vinn@st.usth.edu.vn	1
BI12-014	Chu Nguyen	Chuong	10/20/2003	chuongcn@st.usth.edu.vn	2
BI12-015	Hoa Thinh	Don	12/1/2003	donht@st.usth.edu.vn	2
BI12-016	Tran Van	Si	11/20/2003	sitv@st.usth.edu.vn	2
BI12-017	Trieu Ba	Ky	10/20/2000	kytb@st.usth.edu.vn	2
BI12-018	Dinh Xuan	Anh	9/12/2001	anhdx@st.usth.edu.vn	2
BI12-019	Doan Bao	Tran	12/3/2002	trandb@st.usth.edu.vn	1
BI12-020	Nguyen Tuan	Anh	11/15/2000	anhnt@st.usth.edu.vn	2

Figure 21. R, S, and Student tables.

The below table points out the correspondance with the sample data including R, S, and Student relations (illustrated in Figure 21) which we already discussed in chapter 2.

Operation	Relational Algebra notation	SQL statement

Union	$R \cup S$	`SELECT *`
		`FROM R`
		`UNION`
		`SELECT *`
		`FROM S;`

Intersection	$R \cap S$	`-- Some DBSMSs support INTERSECT`
		`SELECT *`

Operation	Relational Algebra notation	SQL statement

		FROM R
		INTERSECT
		SELECT *
		FROM S;
		-- in MySQL:
		SELECT *
		FROM R
		WHERE 'Attribute A' IN
		  (SELECT 'Attribute A'
		    FROM S);

Different	$R - S$	SELECT *
		FROM R
		WHERE 'Attribute A' NOT IN
		  (SELECT 'Attribute A'
		    FROM S);

Projection	$\pi_{\{ID,FirstName,LastName\}}(Student)$	SELECT ID, FirstName, LastName
		FROM Student

Selection	$\sigma_{Gender=1}(Student)$	SELECT *
		FROM Student
		WHERE Gender = 1;

Renaming	$\rho_{\text{Customer}(ID,Name)}(S)$	SELECT 'Attribute A' ID,
		'Attribute A' Name
		FROM R;

Product	$R \times S$	SELECT *
		FROM R, S;

Theta join	$R \bowtie_{\theta} S$	SELECT *
		FROM R, S
		WHERE $<joining\ condition\ \theta>$;

9. DCL statements

Data Control Language (DCL) consists of commands that control access to data within the database. DCL commands primarily deal with permissions, granting or revoking access rights to users or roles, and ensuring data security and integrity.

Data Control Language commands are crucial in maintaining data security, integrity, and ensuring that access to sensitive information is controlled and managed effectively within a database system.

a. Syntax

Two types of commands are:

- GRANT: The GRANT command allows specific privileges on database objects to be given to users or roles. Privileges can include the ability to SELECT, INSERT, UPDATE, DELETE, or execute procedures and can be applied at various levels (database, table, column, etc.).

```
GRANT SELECT ON table_name TO user_name;
```

- REVOKE: The REVOKE command takes away previously granted permissions from users or roles.

```
REVOKE SELECT ON table_name FROM user_name;
```

Uses of DCL

First, DCL is useful for User Access Control with two tasks:

- User Management: DCL commands are fundamental in managing user access to the database. By granting specific permissions, administrators can control who can view or modify data, execute procedures, or create objects within the database.
- Roles and Privileges: Roles in SQL provide a way to group users and apply permissions collectively. Instead of assigning permissions to individual users, permissions can be granted to roles, simplifying access management and ensuring consistency.

Second, DCL help DBAs to ensure Data Security

- Data Integrity: DCL commands play a vital role in maintaining data integrity by limiting access to sensitive information and preventing unauthorized changes or deletions.
- Compliance and Regulations: Compliance with industry standards and regulations often necessitates strict control over who can access certain types of data. DCL commands enable administrators to enforce these regulations effectively.

b. Best Practices

It is a best practice to grant only the necessary permissions required for users or roles to perform their tasks. This principle, known as "Principle of Least Privilege”, minimizes the risk of unauthorized access or accidental data modification.

Periodic reviews of granted permissions and user access can help identify and rectify any inconsistencies or potential security vulnerabilities.

```
-- Granting SELECT permission to a role
```

```
GRANT SELECT ON employees TO hr_role;

-- Revoking INSERT permission from a user

REVOKE INSERT ON sensitive_table FROM user_name;
```

10. TCL Statements

Transaction Control Language (TCL) in SQL is a subset of commands that manage transactions in a database. Transactions are sequences of operations that are treated as a single unit of work. These operations are typically performed on a database, and they must either all succeed or all fail together, ensuring data consistency and integrity.

TCL commands in SQL consist primarily of three main statements:

- `COMMIT`: The `COMMIT` statement is used to permanently save the changes made during the current transaction. Once a `COMMIT` statement is executed, all changes made by the transaction become permanent and visible to other users or transactions accessing the database. This effectively ends the current transaction and releases any locks or resources held.
- `ROLLBACK`: The `ROLLBACK` statement is used to undo all changes made during the current transaction and restore the database to its state before the transaction began. It effectively cancels the transaction, discarding any changes made since the transaction started.
- `SAVEPOINT`: `SAVEPOINT` allows you to set a point within a transaction to which you can later roll back. It is particularly useful when you want to preserve part of a transaction's work, even if subsequent parts fail and need to be rolled back.

TCL commands are crucial for ensuring data consistency and integrity in database systems. They provide control over the execution of transactions, allowing developers to manage the outcome of operations performed on the database. By properly utilizing TCL commands, developers can ensure that transactions are executed reliably and effectively manage data modifications within the database.

11. PSM Statements

Persistent Stored Modules (PSM) in SQL refer to the ability to store and execute user-defined functions and procedures within the database server. This feature allows developers to define their own functions, procedures, triggers, and making it easier to manage and maintain complex logic and business rules.

By using PSM in MySQL, developers can encapsulate complex logic within the database server, reducing network traffic and improving performance. They also provide a way to enforce data integrity and automate repetitive tasks without the need for external applications.

To create and manage PSM objects in MySQL, you can use SQL statements such as `CREATE PROCEDURE`, `CREATE FUNCTION`, `CREATE TRIGGER`, and `CREATE EVENT`. These objects can be modified or dropped as needed using `ALTER` and `DROP` statements.

PSM in MySQL includes the following components:

- Stored Procedures: These are sets of SQL statements that are stored in the database server and can be called by applications or other SQL statements. Stored procedures can accept input parameters and return output values.
- Stored Functions: Similar to stored procedures, functions are user-defined functions that return a single value. They can be used in SQL queries like built-in functions.
- Triggers: Triggers are special types of stored programs that are automatically executed in response to specific events on a table, such as INSERT, UPDATE, or DELETE operations.
- Events: Events are similar to scheduled tasks that can be defined to run at specific times or intervals. They can be used for tasks such as data maintenance, backups, or reporting.

Overall, Persistent Stored Modules in SQL offer a powerful way to extend the functionality of the database server and enhance the efficiency and reliability of database applications.

Highlight & Summary

This chapter introduces the knowledge and skills needed to leverage SQL effectively for managing and querying relational databases. It starts with an introduction to the significance, origin and history of SQL. After the taxonomy of SQL commands, the syntax and sample statements each types are explained. By mastering the principles and techniques covered in this chapter, readers will be equipped to tackle a wide range of SQL challenges and become proficient in leveraging SQL for effective database management and querying.

Exercises

SQL Theory

1. What is SQL, and what does it stand for?
2. What are the main categories of SQL commands? Briefly explain each category.

3. Explain the difference in the objectives between the SQL commands SELECT, INSERT, UPDATE, and DELETE.
4. How many clauses does the SELECT statement comprise of?
5. Describe the purpose of the WHERE clause in SQL queries.
6. Explain the difference between INNER JOIN, LEFT JOIN, and RIGHT JOIN in SQL.
7. What are the transaction control commands in SQL, and why are they important?
8. What is a subquery, and how is it used in SQL?
9. Explain the difference between GROUP BY and ORDER BY clauses in SQL.

Lite AMS database

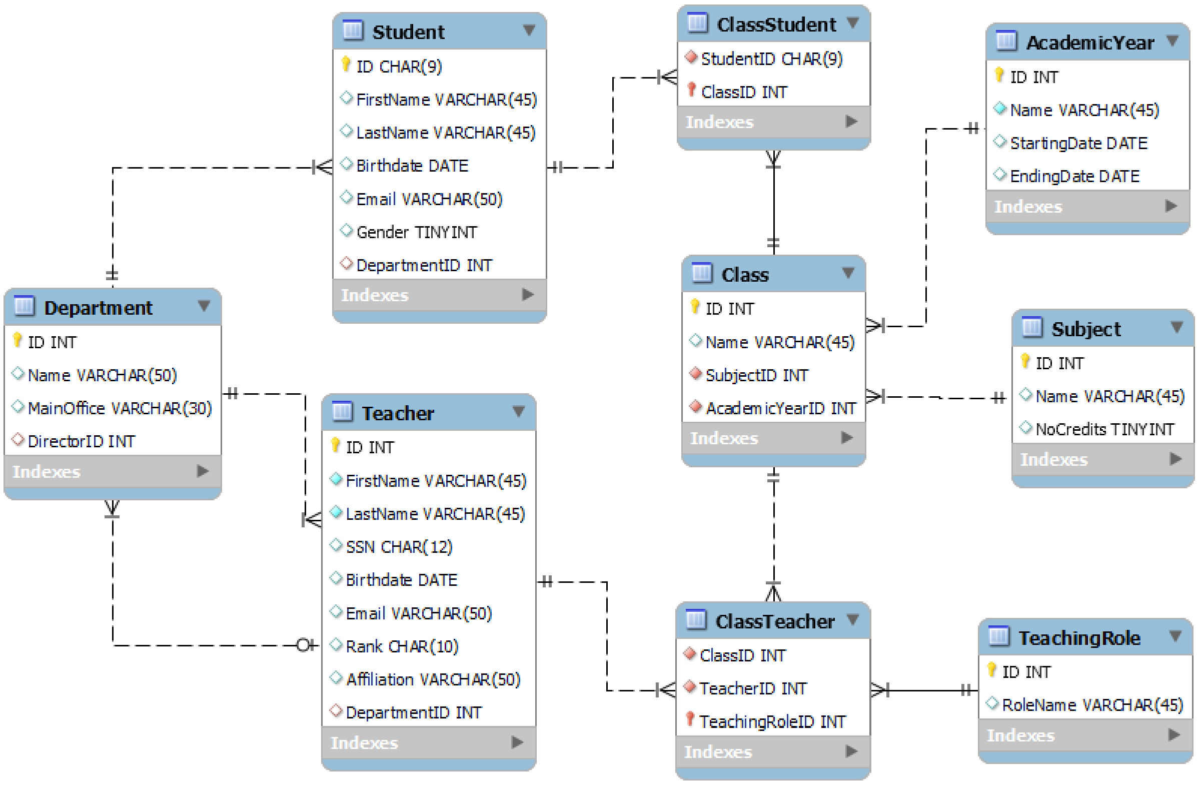


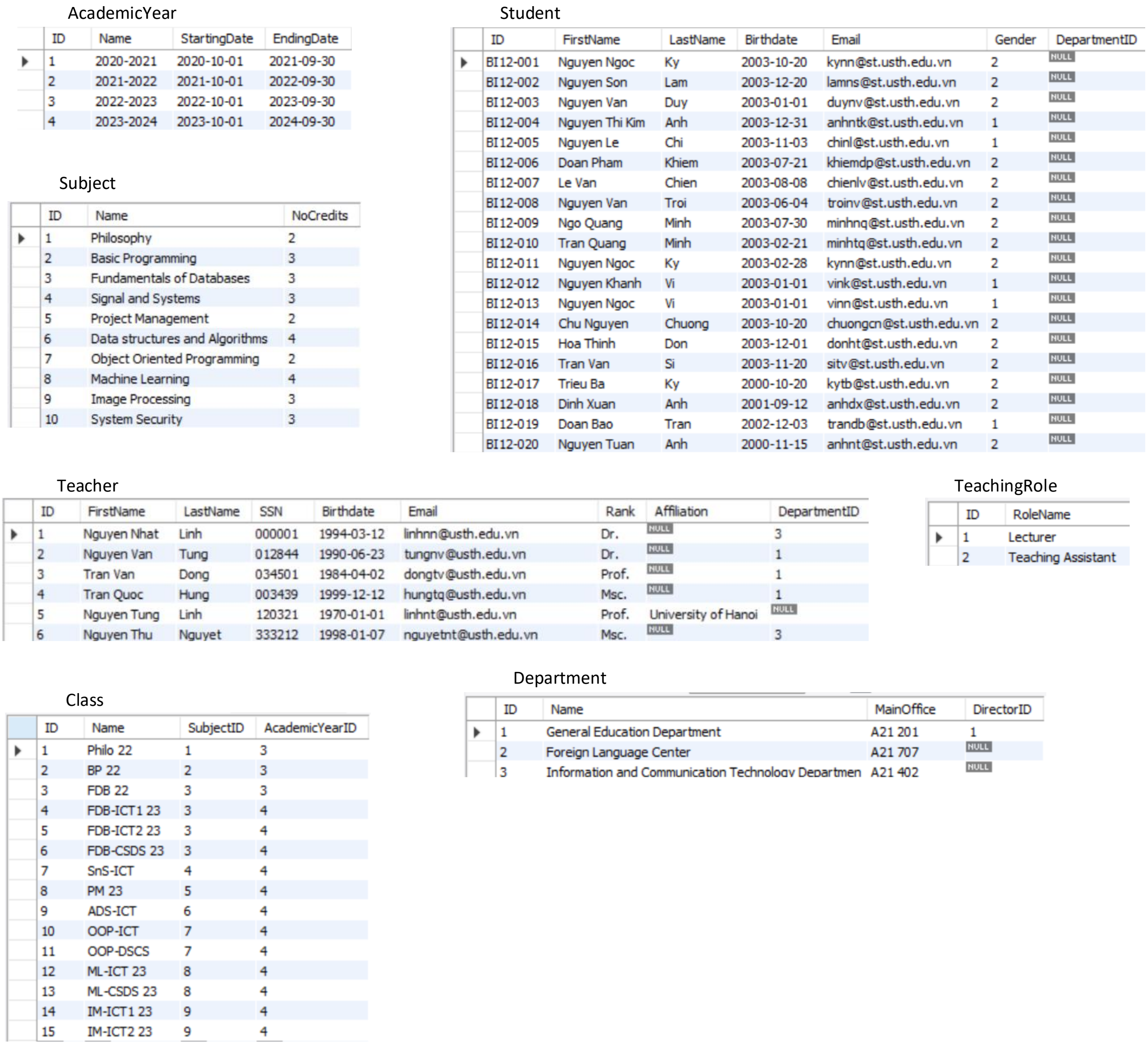
Figure 22. The design of Lite AMS database.

7. Write a SQL statement to create a new database named “Lite AMS”.
8. Write SQL statements to define tables of Lite AMS database with Primary and Foreign Keys illustrated by Figure 22. Notably, there is a recursive dependency between Department and Teacher representing by 02 foreign keys: Teacher.DepartmentID refers to DepartmentID and Department.DirectorID refers to TeacherID. We cannot define these tables with full foreign keys

directly. Instead, one of these foreign keys must be added later by ALTER TABLE statement.

9. Write SQL statements to insert data for Lite AMS”. The sample data in Figure 23 can be a hint.
10. Write SQL queries to:

    a. Find female students whose first name staring with ‘A’.
    b. Show the list of Subject names, the corresponding classes and Academic year.
    c. Show list of student in each class. The result includes all classes even when there are classes without any registered student.
    d. Show list of class with their number of registered students and sort the result by number of student in descending order. The results include classes having no student.
    e. Show list of class with their number of registered students and sort the result by number of student in descending order. The results include classes having more than 9 students.
    f. Show list of teachers with their roles for each class.
    g. Show the youngest student with 4 columns: ID, FullName (FirstName + ‘ ‘ + Lastname), Email, and CurrentAge.
    h. Show list of Teacher with following columns: ID, FullName, Department Name, Director’s FullName.
    i. Show list of teachers who have not been assigned to any class.
    j. Show list of subjects having more than 1 class in an academic year. The resulting columns include: SubjectName, Number of class, Academic year.



AcademicYear

ID	Name	StartingDate	EndingDate
1	2020-2021	2020-10-01	2021-09-30
2	2021-2022	2021-10-01	2022-09-30
3	2022-2023	2022-10-01	2023-09-30
4	2023-2024	2023-10-01	2024-09-30

Student

ID	FirstName	LastName	Birthdate	Email	Gender	DepartmentID
BI12-001	Nguyen Ngoc	Ky	2003-10-20	kynn@st.usth.edu.vn	2	NULL
BI12-002	Nguyen Son	Lam	2003-12-20	lamns@st.usth.edu.vn	2	NULL
BI12-003	Nguyen Van	Duy	2003-01-01	duynv@st.usth.edu.vn	2	NULL
BI12-004	Nguyen Thi Kim	Anh	2003-12-31	anhntk@st.usth.edu.vn	1	NULL
BI12-005	Nguyen Le	Chi	2003-11-03	chinl@st.usth.edu.vn	1	NULL
BI12-006	Doan Pham	Khiem	2003-07-21	khiemdp@st.usth.edu.vn	2	NULL
BI12-007	Le Van	Chien	2003-08-08	chienlv@st.usth.edu.vn	2	NULL
BI12-008	Nguyen Van	Troi	2003-06-04	troinv@st.usth.edu.vn	2	NULL
BI12-009	Ngo Quang	Minh	2003-07-30	minhnq@st.usth.edu.vn	2	NULL
BI12-010	Tran Quang	Minh	2003-02-21	minhtq@st.usth.edu.vn	2	NULL
BI12-011	Nguyen Ngoc	Ky	2003-02-28	kynn@st.usth.edu.vn	2	NULL
BI12-012	Nguyen Khanh	Vi	2003-01-01	vink@st.usth.edu.vn	1	NULL
BI12-013	Nguyen Ngoc	Vi	2003-01-01	vinn@st.usth.edu.vn	1	NULL
BI12-014	Chu Nguyen	Chuong	2003-10-20	chuongcn@st.usth.edu.vn	2	NULL
BI12-015	Hoa Thinh	Don	2003-12-01	donht@st.usth.edu.vn	2	NULL
BI12-016	Tran Van	Si	2003-11-20	sitv@st.usth.edu.vn	2	NULL
BI12-017	Trieu Ba	Ky	2000-10-20	kytb@st.usth.edu.vn	2	NULL
BI12-018	Dinh Xuan	Anh	2001-09-12	anhdx@st.usth.edu.vn	2	NULL
BI12-019	Doan Bao	Tran	2002-12-03	trandb@st.usth.edu.vn	1	NULL
BI12-020	Nguyen Tuan	Anh	2000-11-15	anhnt@st.usth.edu.vn	2	NULL

Subject

ID	Name	NoCredits
1	Philosophy	2
2	Basic Programming	3
3	Fundamentals of Databases	3
4	Signal and Systems	3
5	Project Management	2
6	Data structures and Algorithms	4
7	Object Oriented Programming	2
8	Machine Learning	4
9	Image Processing	3
10	System Security	3

Teacher

ID	FirstName	LastName	SSN	Birthdate	Email	Rank	Affiliation	DepartmentID
1	Nguyen Nhat	Linh	000001	1994-03-12	linhnn@usth.edu.vn	Dr.	NULL	3
2	Nguyen Van	Tung	012844	1990-06-23	tungnv@usth.edu.vn	Dr.	NULL	1
3	Tran Van	Dong	034501	1984-04-02	dongtv@usth.edu.vn	Prof.	NULL	1
4	Tran Quoc	Hung	003439	1999-12-12	hungtq@usth.edu.vn	Msc.	NULL	1
5	Nguyen Tung	Linh	120321	1970-01-01	linhnt@usth.edu.vn	Prof.	University of Hanoi	NULL
6	Nguyen Thu	Nguyet	333212	1998-01-07	nguyetnt@usth.edu.vn	Msc.	NULL	3

TeachingRole

ID	RoleName
1	Lecturer
2	Teaching Assistant

Class

ID	Name	SubjectID	AcademicYearID
1	Philo 22	1	3
2	BP 22	2	3
3	FDB 22	3	3
4	FDB-ICT1 23	3	4
5	FDB-ICT2 23	3	4
6	FDB-CSDS 23	3	4
7	SnS-ICT	4	4
8	PM 23	5	4
9	ADS-ICT	6	4
10	OOP-ICT	7	4
11	OOP-DSCS	7	4
12	ML-ICT 23	8	4
13	ML-CSDS 23	8	4
14	IM-ICT1 23	9	4
15	IM-ICT2 23	9	4

Department

ID	Name	MainOffice	DirectorID
1	General Education Department	A21 201	1
2	Foreign Language Center	A21 707	NULL
3	Information and Communication Technology Departmen	A21 402	NULL

Figure 23. Sample data of some table in the Lite AMS Database.

Northwind database

11. On the Northwind database (Figure 24), write SQL queries to:

    a. Retrieve the top 5 cheapest products.

    b. Show the minimum, maximum, and average price in Products table.

    c. Show the minimum, maximum, and average price of each category.

    d. Use a subquey, show Product list (name, list_price) expensive than the average price of its category.

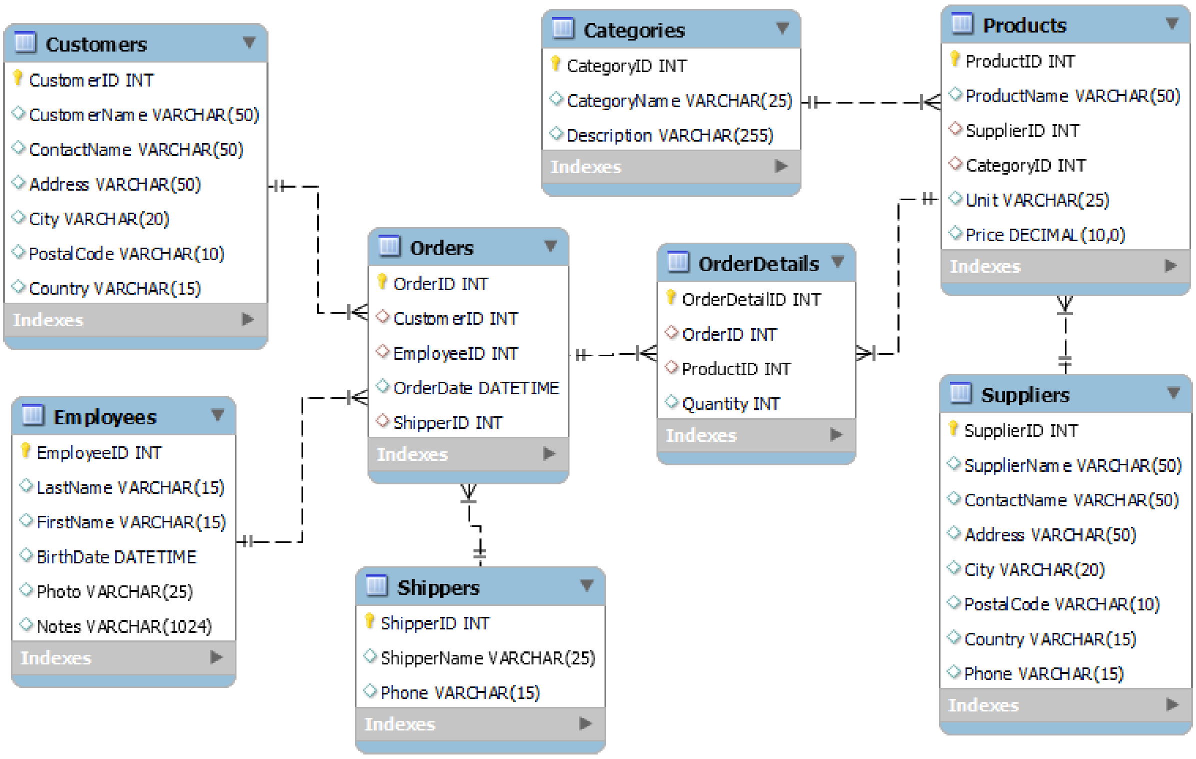


Figure 24. The design of Northwind database.

e. List all orders made after ‘1996-09-05’

f. Show ProductName, unit price, quantity, value of orders details whose order_id = 10251. Note: value can be calculated as price * quantity.

g. Write a query to show information of each order including OrderID, OrderDate, Customer’s ContactName, TotalValue of the order orders made after ‘1996-09-05’.

h. As the company will give rewards for employees who have total value of all order more than 30000$, the director needs a report listing these employees (FullName, sale in dolars) from high to low. Write a query for this report.

i. Create a query showing all partners (customers and suppliers) of Northwind. The columns consist of Name (CustomerName or SupplierName), Addess, Type (‘C‘ for customers, ‘S’ for suppliers).

j. Show list of customers who have not made any order.

Chapter 4: Non-table Objects in RDBMS

The databases in RDMBS consist of not only data tables but also some additional objects to enhance the efficiency, organization, and automation of data operations. This chapter discusses how to speed up queries by using INDEX, enhance the convenience and security of writing queries with VIEW, and obtain several advantages by using programmable objects including STORED PROCEDURE, and TRIGGER.

By the end of the chapter, readers should achieve the following goals:

- Understand the concept and uses of index, view, store procedure, and trigger in RDMBS.
- Able to write SQL statements to define and manage these objects.

1. Index

Index in RDBMs is an essential tool for optimizing query performance and speeding up data retrieval. It is akin to the index found in a book. It is a structured data object that improves the speed of data retrieval operations on a table at the cost of additional storage space and some overhead during data modification. When a query is executed, DBMS can utilize indexes to swiftly locate the relevant rows instead of scanning the entire table.

Because the ISO SQL standards do not cover performance concerns, there is not a universally standardized SQL syntax for indexes across different Relational Database Management Systems (RDBMSs).

a. Types of Indexes

To classify indexes, we first need study data structures that RDMBS employ to store tables. Some DBMSs such as SQL Server and PostgreSQL allow store data in no particular order, meaning the rows are simply placed into the pages as they arrive. That data structure is referred to as heap. A heap is composed of multiple pages linked together in a doubly linked list. Each page in SQL Server is 8 KB in size. Heaps offer efficient way to store data without the overhead of maintaining the order of rows. They are particularly useful for scenarios requiring fast insert performance or when dealing with temporary or staging data. On the other hand, all DBMSs let users to opt for the order of data according to a set of columns. Therefore, we can either use a clustered index as the location to store data table, or build separated indexes (Non-clustered indexes and Full-text search) which contain only pointer or address of data. The details of 03 main types of indexes are described below:

- **Clustered index** is the index that physically arranges the data in a table based on the values of the clustered index key. The rows in the table are stored on disk in the same order as the clustered index key. Unlike other indexes, which create a separate data structure, a clustered index directly affects the physical storage of the data. The rows are organized by these row IDs, which MySQL assigns. Consequently, the rows are physically arranged in the order they were inserted. Clustered indexes are generally faster for retrieving ranges of data because the data is stored sequentially on the disk. In MySQL, when the PRIMARY KEY on a table is defined, MySQL uses it as the clustered index so the PRIMARY KEY is always clustered. A primary key should be defined for each table. If a table lacks a PRIMARY KEY or an appropriate UNIQUE index, MySQL creates a hidden clustered index called GEN_CLUST_INDEX on a synthetic column containing row ID values.

- **Non-clustered indexes** do not alter the physical order of the data in the table. Instead, they create a separate structure that stores the index key values along with pointers to the actual data rows. A table can have multiple non-clustered indexes, which makes them highly versatile for optimizing a variety of queries. If the UNIQUE is chosen in the index definition, the index is similar to a primary key index; it enforces uniqueness but allows NULL values (one per table). The syntax to define a non-clustered index in MySQL is:

```
CREATE [UNIQUE] INDEX index_name

ON table_name (column1, column2, ...);
```

- **Full-text search** is a powerful feature that enables sophisticated and efficient searching of textual data within database columns. Unlike traditional indexing methods that rely on exact matches, full-text search leverages specialized indexing techniques to handle complex queries involving keywords, phrases, and patterns. This approach is particularly useful for searching large volumes of unstructured or semi-structured text, such as documents, articles, or product descriptions. The primary data structure used for full-text search is the **inverted index**, also known as an **inverted file** or **posting list**. This structure is designed to efficiently handle queries involving keywords and phrases across large volumes of text. Here's how it works and its components. This capability enhances the user experience by providing more accurate and nuanced search results, making it an essential tool for applications that require robust text-searching functionality.

b. Benefits of Indexes

Indexes in RDBMSs offer a multitude of advantages. Firstly, they significantly enhance query performance by expediting data retrieval operations. With indexes in place, the system can swiftly pinpoint the required rows, reducing the need for full table scans. Additionally, indexes facilitate faster sorting and grouping of data, optimizing operations that involve ordering or categorizing information. Moreover, they play a pivotal role in improving join performance, particularly in complex queries involving multiple tables, by streamlining the process of matching and merging data. Overall, indexes substantially contribute to overall system efficiency, enabling quicker data access and manipulation, which is integral to the performance of RDBMSs in various applications and scenarios.

c. Considerations and Best Practices

While indexes offer substantial performance gains, using them carefully is necessary. The following points should be considered with the use of indexes.

- Balancing Act: Over-indexing can lead to increased storage requirements and slower write operations. Striking a balance between read and write performance is essential.
- Index Selectivity: Choosing the right columns to index is crucial. Highly selective columns (those with a wide range of distinct values) are optimal candidates for indexing.
- Regular Maintenance: Periodically check and optimize indexes. Unused or redundant indexes can hamper performance.
- Understanding Query Execution Plans: Use MySQL's EXPLAIN statement to analyze how queries are executed and identify potential areas for index optimization.

2. View

In a RDBMS, a view is a virtual table derived from one or more tables or other views as illustrated in Figure 25. It does not store the data itself but rather presents a tailored perspective of the data that exists in the database. Views does not require additional storage except the source code defining it.

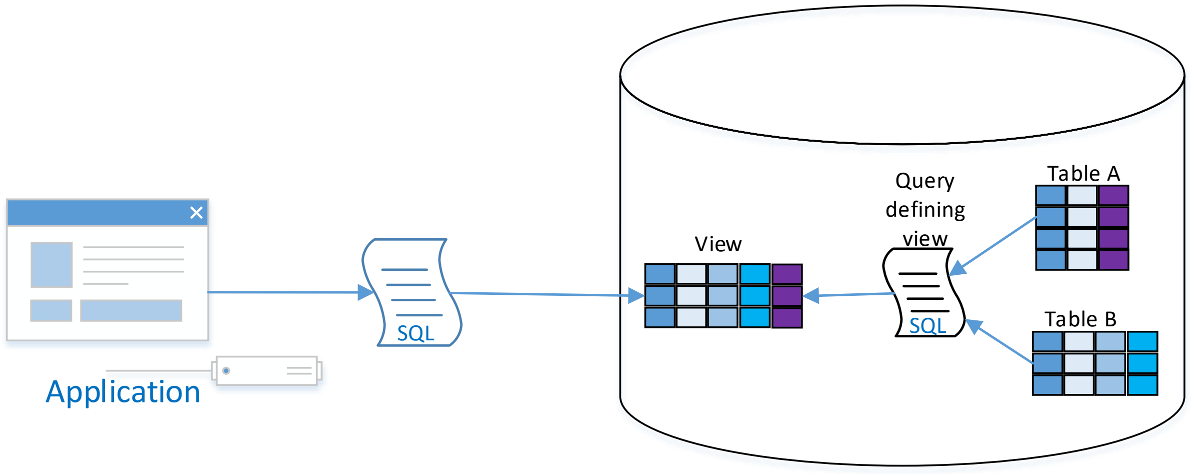


Figure 25. View in a database.

In constrast of ordinary views, a materialized view is a database object that contains the results of a query and stores them physically, unlike a regular view, which only defines the query dynamically. They are used to enhance performance by pre-computing and storing complex query results, which can then be quickly accessed without re-executing the original query each time. However, because materialized views store data, they require regular maintenance and refreshing to ensure the data remains up-to-date with the underlying base tables.

The remaining of this section focuses on ordinary views. Discussions and a survey on materialized views can be found at [6].

a. *SQL syntax for Views*

The SQL syntax to create a view in My SQL is below:

```
CREATE VIEW view_name AS

A_select_statement;
```

The syntax to remove a view is really simple, like other object removal:

```
DROP VIEW view_name;
```

b. *Benefits*

Using view can provide the following benefits and advantages:

- Simplification of Queries: Views allow encapsulating complex SQL logic into a single entity. For instance, a view can join multiple tables, apply filters, or perform calculations. By abstracting this complexity, users can query the view without needing to understand or rewrite the intricate underlying logic each time. Moreover, views enable users to focus on specific subsets of data, i.e. instead of dealing with entire tables, views present a customized view of the data—selected.
- Backward capability: Views contribute to maintaining the backward compatibility of tables by serving as an intermediary layer that shields users and applications from structural changes in the underlying tables. For example, when the alterations are made to underlying tables, a view can be created or re-defined so that users can keep the statement to perform the same task by working with the view instead of the already-changed underlying tables.
- Security and Access Control: Views can enforce security measures by restricting access to sensitive data. They allow administrators to grant users access to a view containing only the necessary columns or rows while hiding the underlying tables' complexities. This simplifies queries by providing a controlled and sanitized view of the data.

c. *Updatable Views and Non-Updatable Views*

In MySQL, an updatable view is a view that allows certain modifications to the data underlying the view. By default, not all views are inherently updatable. There are specific criteria and limitations that determine whether a view can be updated directly.

Criteria for Updateable Views in MySQL:

- Single Table View: Generally, an updatable view in MySQL involves selecting columns from a single underlying table. Views created from multiple tables, views with aggregations, sub queries, joins, or derived columns, might not be directly updatable.
- Simple SELECT and WHERE Clauses: The SELECT statement in the view definition should not contain complex operations like sub queries, aggregate functions, or derived columns. Additionally, the WHERE clause should be simple and refer to columns from a single table.
- No DISTINCT, UNION, or GROUP BY: Views using DISTINCT, UNION, or GROUP BY clauses are typically not updatable in MySQL.
- No Aliasing or Calculations: Views with calculated columns, aliasing columns, or involving expressions are generally not directly updatable.

For example, consider the table Teacher(ID, FirstName, LastName, SSN, Birthdate, Email, Rank, Affiliation, DepartmentID). Two of its columns (SSN, BirthDate) are sensitive to the public so you can define a updatable view consisting of tenured teachers with un-sensitive information as follows:

```
CREATE VIEW TenuereTeacher AS
SELECT FirstName, LastName, Email, Rank, DepartmentID
FROM Teacher
WHERE DepartmentID IS NOT NULL;
```

In contrast, the following view is un-updatable because of a derived columns titled FullName:

```
CREATE VIEW TenuereTeacherWithFullName AS
SELECT CAST(FirstName,' ', LastName), Email, Rank, DepartmentID
FROM Teacher
WHERE DepartmentID IS NOT NULL;
```

3. Stored procedure

Stored procedures in a database management system (DBMS) are sets of precompiled SQL statements stored as a named object within the database. They encapsulate and execute a sequence of one or more SQL statements and procedural logic, offering numerous benefits for database management and application development. The way a stored procedure works is illustrated by Figure 26.

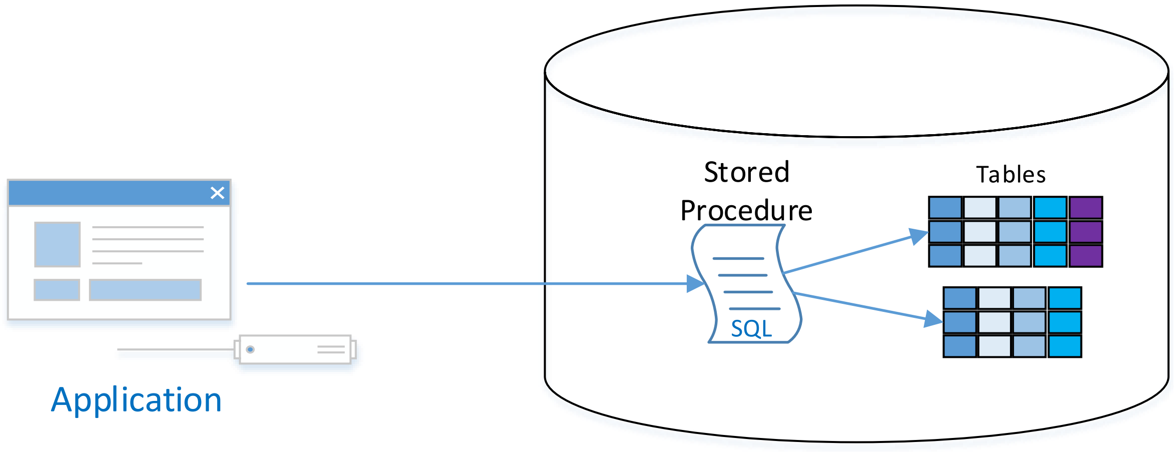


Figure 26. Stored procedure in a database.

a. Advantages

Stored procedures offer a multitude of benefits in database management and application development. Here are some key advantages:

- Improved Performance: Stored procedures are precompiled, optimized for faster execution, resulting in improved performance compared to executing individual SQL queries. Stored procedures can reduce the network traffic as they are precompiled and stored in the database, meaning the source code of statements which is normally much longer than the name and parameters of procedures is not transferred via the network.
- Enhanced Security: They allow controlled access to database functionalities without granting direct table access. Users interact with the database through the interface of stored procedures, enforcing security measures. Moreover, using parameterized queries within stored procedures helps prevent SQL injection attacks, enhancing overall security.
- Improved reusability and modularity: Once created, stored procedures can be used multiple times across various applications or parts of the same application, promoting code reuse and reducing redundancy. They facilitate a modular approach to database design, making it easier to maintain and update database logic. That can simplify the maintenance of SQL statements. Changes made to a stored procedure are reflected in all instances where it's called, reducing the need for modifications across multiple applications or queries. Debugging and troubleshooting become easier as stored procedures isolate database logic in a structured manner.

b. Syntax

The general syntax to create a Stored Procedure in My SQL is:

```
CREATE
[DEFINER = user]
PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
```

It is important to set the appropriate delimiter before and after the procedure definition, as shown with `DELIMITER //` and `DELIMITER ;`. Adjust the SQL statements within the `BEGIN...END` block according to your desired logic for the stored procedure.

```
DELIMITER //
CREATE PROCEDURE procedure_name (parameter_list)
BEGIN
    [characteristics]
    SQL statements
END//
DELIMITER ;

DELIMITER ;
```

For example, to define a procedure to show a list of teachers of a department, we run:

```
DELIMITER //
CREATE PROCEDURE GetTeacher(DepID INT)
BEGIN
SELECT * FROM
    Teacher
    WHERE DepartmentID = DepID;
END//
DELIMITER ;
```

To call this procedure, simply write:

```
CALL GetTeacher(3);
```

4. Trigger

Triggers can be considered as a special type of stored procedure in RDBMS. They are designed to automatically execute a set of predefined actions in response to specific events occurring in a relational database management system (RDBMS). These events can encompass various database operations, such as `INSERT`, `UPDATE`, `DELETE`, or specific data manipulation actions performed on specific tables or views.

Triggers serve multiple purposes in a database environment. They enable the automation of routine tasks, enforce data integrity constraints, enhance security measures, and facilitate complex data processing within the database itself.

The syntax of defining a trigger in MySQL is as follows:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
    trigger_time trigger_event
ON tbl_name FOR EACH ROW
[trigger_order]
trigger_body
```

Where:

```
trigger_time: { BEFORE | AFTER }
trigger_event: { INSERT | UPDATE | DELETE }
trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

For example, given a table Subject (ID, Name, NoCredits), we can define a trigger to prevent someone from updating the NoCredits into a negative value as below:

```
DELIMITER //
CREATE TRIGGER PreventUpdateNegativeCredits
BEFORE UPDATE ON Subject
FOR EACH ROW
BEGIN
    IF NEW.NoCredits < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot update negative NoCredits';
    END IF;
END //
DELIMITER ;
```

Similarly, a trigger named `PreventInsertNegativeCredits` will prevent inserting new records with negative values for NoCredits field.

```
DELIMITER //
CREATE TRIGGER PreventInsertNegativeCredits
BEFORE INSERT ON Subject
FOR EACH ROW
BEGIN
    IF NEW.NoCredits < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot insert negative NoCredits';
    END IF;
END //
DELIMITER ;
```

Highlight & Summary

In this chapter, we delve into the some additional objects in RDMBS: indexes, views, stored procedures, and triggers. These components play pivotal roles in enhancing data retrieval, manipulation, and overall system performance. Throughout this chapter, real-world examples, best practices, and practical tips are provided to help readers grasp the concepts and apply them effectively in their database development projects. By understanding the roles and capabilities of indexes, views, stored procedures, and triggers, readers will be equipped to optimize database performance, simplify application development, and maintain data integrity in their database environments.

Exercises

1. What is the purpose of an index in a database?
2. Describe the difference between clustered and non-clustered indexes.
3. How do indexes improve query performance? Provide examples.
4. What is a view in a database, and how does it differ from a table?
5. Describe some scenarios where views might be useful in database applications.
6. What are the benefits of using views for security enforcement?
7. Can you update data through a view?
8. What is a stored procedure and what are its purposes in a database?
9. What are the advantages of using stored procedures over ad-hoc SQL queries?
10. How do you pass parameters to a stored procedure, and what are the benefits of parameterized queries?
11. What is a trigger, and how does it differ from a stored procedure?
12. Describe scenarios where triggers might be useful in maintaining data integrity.
13. What are some potential risks or pitfalls associated with using triggers in a database?
14. Write SQL statements to test and demonstrate the effects of two triggers (`PreventUpdateNegativeCredits` and `PreventInsertNegativeCredits`) in section 4.

Chapter 5: Relational Database Analysis and Design

In the preceding chapters, you have acquired essential concepts and techniques for working with pre-designed relational databases. However, you might find it challenging to apply these skills and knowledge to address specific business problems, like managing academic affairs at a university or organizing a library's book inventory, because there may not exist a fitting-well database for that problem in advance. Designing a database properly is crucial to establishing a database meeting the needs of users. The efficiency and complexity of queries depend directly on how the database is structured.

This chapter presents a typical process for designing a relational database for management software, using the Academic Management System as a case study. The details techniques of each step are discussed and illustrated with the example of academic management. By the end of the chapter, readers should achieve the following goals:

- Get to know the overall process of obtaining a database design from scratch.
- Learn in detail how to analyze the data aspect of a business by using Entity Relationship Diagrams (ERD) to describe the data model.
- Get to know the principals and best practices to transform an ERD into a relational database design.

1. Database development processes

Databases are always the core component of any information system; therefore, the database development processes should be discussed in the context of the System development life cycle (SDLC) process. SDLC is a structured methodology used in software development and IT project management. It outlines a series of phases or steps that guide the creation, implementation, and maintenance of information systems or software applications. This systematic approach ensures that projects are well-planned, executed efficiently, and meet the specified requirements and objectives.

The SDLC typically consists of several key phases, although the exact steps and their names might vary depending on the methodology adopted:

- System Definition: This initial phase involves defining project scope, goals, and requirements. It includes feasibility studies, cost-benefit analyses, and outlining resources and timelines.
- Analysis: In this phase, database developers work closely with stakeholders to understand data requirements. They identify what information needs to be stored, how it will be accessed, and any specific functionalities required. The most important output for database analysis is the data model of business, which is an abstract representation of the data structure. Techniques like Entity-Relationship Diagrams (ERDs) help visualize relationships between different data entities.
- Design: Based on the gathered requirements, developers create the database schema, specifying tables, fields, relationships, constraints, and indexes. This step includes normalizing the data to ensure efficient storage and minimize redundancy.
- Implementation: Here, the actual development of the system takes place. Programmers write code, databases are created, and the system is built according to the design specifications. There are two tasks relating to databases. First, in database creation, the actual database is built based on the designed schema using SQL (mainly DDL statements). Second, the data Population step then populates the database with sample or initial data to ensure it aligns with the application’s needs for testing purposes.
- Testing: The developed system undergoes rigorous testing to identify and fix any defects or issues. This phase ensures that the system meets quality standards and functions as intended. Two kinds of database testing need to be performed. Data Integrity Testing ensures that data is stored correctly, relationships are maintained, and constraints are enforced. On the other hand, Performance Testing assesses the database’s performance under various conditions to ensure it meets expected response times and can handle expected loads.

- Deployment: The system is deployed or released for use by the end-users. This phase might involve training, data migration, and transitioning from older systems to the new ones. The developed database is integrated with the software application.
- Maintenance: Regular monitoring, updates, and optimization of the database to ensure it continues to perform optimally and meets changing requirements.

Database development within the SDLC emphasizes the importance of aligning the database structure with the software application’s needs. It aims to create a secure, efficient, and scalable data storage solution that supports the application’s functionalities and evolves with changing business requirements. Collaboration between developers, database administrators, and stakeholders is vital at every stage to ensure a successful database development process.

Throughout the SDLC, iterations and feedback loops may occur, allowing for adjustments based on changing requirements or unforeseen challenges. Various methodologies, such as Agile, Waterfall, or Iterative models, offer different approaches to the SDLC, each with its strengths and suitability for different project types.

An information system is a comprehensive framework composed of people, processes, data, and technology that work together to gather, process, store, and disseminate information. It's a structured system designed to collect, manage, and distribute data in various forms, transforming raw data into valuable insights that support decision-making, facilitate operations, and enable communication within an organization or across multiple entities.

Analyzing and designing a database is a crucial step in creating a robust and efficient data management system. It involves a meticulous process of understanding, organizing, and structuring data to meet the specific needs of an organization or a project. This process encompasses various stages, starting from comprehending the requirements and objectives to constructing a blueprint that outlines the database structure.

2. System Definition

Every database project begins with a clear comprehension of the problem statement, typically outlined in a project plan document created during the initial system definition phase. Among sections in the project plan, the project goals and scope drastically decide the database aspect in upcoming steps.

The goals and scope of an Academic Management System (AMS) can be comprehensive, aiming to streamline various academic processes and enhance efficiency within educational institutions. Here are some goals of the AMS:

- Efficient Information Management: Centralize student, faculty, and course information for easy access and management.
- Enhanced Communication: Facilitate effective communication between students, faculty, and administration.
- Student Progress Tracking: Monitor and track student performance, grades, attendance, and overall progress.
- Resource Allocation: Optimize resource allocation, including classrooms, and faculty.
- Administrative Efficiency: Simplify administrative tasks such as admissions and record-keeping.
- Data Security and Privacy: Ensure data security measures to protect sensitive information.

And the sample scope of the AMS:

- User Access and Security: Implement user roles, access controls, and data encryption for security
- Student Information Management: Capture and maintain student profiles, enrollment details, academic records, etc.
- Faculty Information Management: Store faculty profiles, qualifications, schedules, and performance evaluations.
- Course Management: Facilitate course creation, scheduling, allocation of resources, and curriculum updates.
- Attendance and Grading: Automate attendance tracking and grading systems, providing real-time updates.
- Reporting and Analytics: Generate reports on student performance, faculty workload, and other key metrics.

The scope may vary based on the specific requirements of the institution and the functionalities required in the AMS. This system aims to automate and streamline various administrative and academic processes, fostering a more efficient and organized educational environment.

3. User Requirement Collection

The user requirement collections step takes the project goals and scopes as the inputs and results in the approved software requirement specification as its output.

To ensure the requirements actually meet the needs of intended users, the following activities should be carried out:

- Stakeholder Identification: to identify who is involved in using or interacting with the database and their roles, responsibilities, and requirements concerning data access, manipulation, and reporting.
- Interviews and Workshops

There are two requirement types: non-functional and functional. The former includes some aspects, such as security, maintainability, portability, and scalability, which do not directly affect the database scheme design; we, therefore, skip it in the scope of this course. The latter, in contrast, will decide how we design a database. A sample **functional requirement description** of AMS can be summarized by the use case diagram in Figure 27 and is detailed as follows:

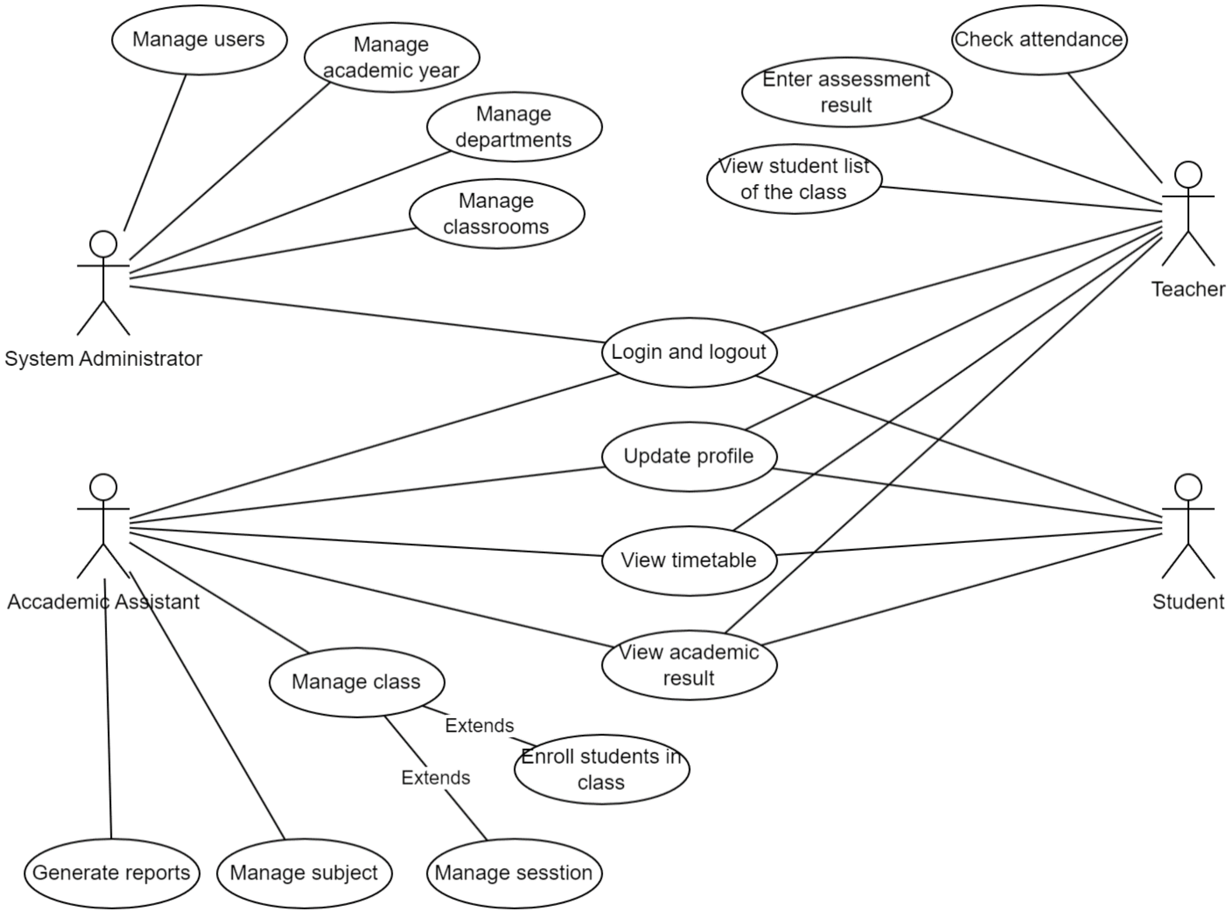


Figure 27. The use case diagram of AMS.

- Authenticaion (**Login and Logout**). The AMS will allow access to only authorized users with their roles (system administrator, academic assistant, teacher, and student). The login function is the entrance point facilitating access to the system. The login screen contains a username and a password field with a mask to hide the typing character. When the user enters the correct username and password, he/she will be authenticated and then can access relevant modules. Users can log out of the system when they want to finish a working session. Besides students, any user can update their profile to ensure that their personal data (email, phone number) is always up-to-date.

- **Manage user**: the system allows the administrator to create new users with username and password, maintain their information (social security number (SSN), first name, last name, birthdate, gender, email, phone number) remove, and disable he/she. The additional information of a teacher include rank (e.g. Prof., Assoc. Prof., Ph.D., Msc), affiliation (if he/she is a conjunction lecturer), department (if he/she is an long-term staff). The additional information of a student is the department he/she belongs to.
- **Manage academic year** is used by the system administrators. Each academic year includes name, starting, and ending date.
- **Manage department**. The system administrators should be able to manage the list of departments with the name and main office. In addition, there should be a tool to define which department that a teacher or student belongs to. A department has a director who is one of its teachers.
- **Manage classroom**. The system administrator can manage the list of classrooms with their name, capacity (how many seats), and description. A classroom locates in a building which has name. The classroom name is unique within a building but not globally.
- **Manage subject** is used by the academic assistants. The data includes the name and number of credits.
- Manage class. The academic assistants can define a class as the implementation of a subject for a group of students. A student can follow multiple classes. Each class can select a student in the list to be the monitor. The class has its name, the academic year it belongs to, the selected student as the monitor, and the teachers with their roles (main lecturer, invited lecturer, or teaching assistant). Each class is planned in several sessions (name, main teacher, at which classroom, starting time, ending time, and session type (lecture, practice)). A teacher or student cannot participate in more than one session at the same time.
- **View timetable**: this functionality displays sessions of classes in the calendar by weeks or months. The timetable of any class can be viewed by any user; however, teachers and students, by default, can see only sessions relating to them directly.
- Teachers can **View student list of the class,** and **Check attendance (**mark the state of each student in each session), and **Enter assessment results** for students. Each time a student is evaluated, the assessment date is also recorded.
- **View academic result**: the academic results can be viewed by only relating users, meaning a student can see only his/her results, a teacher can see the grade

book of the whole class, and an assistant can show the results of managed classes.

- **Generate reports**: reports are generally generated for assistants. Some important ones include the academic results of specific students, the list of students has not passed a subject, the total teaching hours of teachers (detailed in each subject, each session type), the top n (or n percent) students with highest GPA in each academic year.

4. Data Analysis using ERD

Obtaining the functional requirement introduced in the previous section can help us draw up a list of the software functionalities and data; however, it is insufficient to design a database immediately as we have not understood sufficiently the structure and relationship of data. We need, therefore, to perform another step to analyze the model of all the data we need to manage. There exist various techniques to describe the data model; however, the most common one is the Entity-Relationship Model (ERD) created by Peter Chen [5] quite early in 1976. ERDs serve as a tool to depict graphically the structure of data using 3 principal element types, including entity set, relationship, and attributes. ERDs can facilitate communication between stakeholders, aiding in the visualization and understanding of the database schema's structure and dynamics. In the scope of this book, we consider ERD as the output of the relational database analysis as it is a good input document for the database design step.

a. Entity Set and Attribute

The entity term refers to a distinct object, concept, or thing within a database. An entity set, on the other hand, refers to a collection of similar entities. Entity sets help organize and categorize related entities, facilitating the conceptualization and design of a database schema. As the goal of using a database is to manage data of entities, each entity set always consists of their attributes which implies their properties or characteristics. The attributes in databases are simple values, so normally in built-in and primitive data types such as numeric types (int, float…), character, string, and text but not in complex types like struct, list, set. For instance, in AMS, an entity can be a specific department such as "Information and Communication" while the corresponding entity set is the "Department" with attributes like ID, Name, Main Office. In ERD, an entity set is represented by a rectangle and its attributes are ovals connecting to the rectangle by a solid line while the key attributes are underlined (see Figure 28).

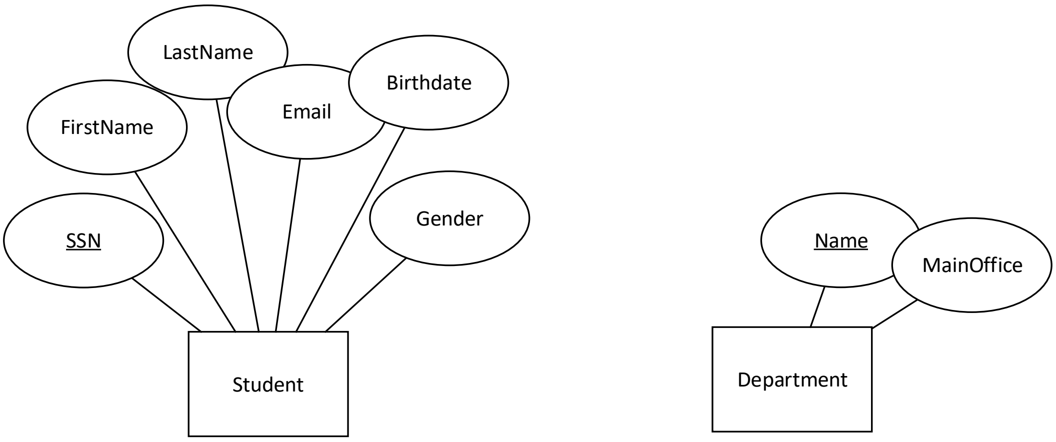


Figure 28. Entity set and attributes.

b. Relationship

In ERDs, relationships define how entities are connected or associated with each other within a database. These connections are crucial for modeling the interactions and dependencies between different entities. A relationship is represented by a diamond shape and lines connecting entity sets. We may optionally use **cardinality** to indicate the number of instances of one entity set that can be related to instances of another. For example, as illustrated in Figure 29, according to the functional requirement descriptions, students belonging to departments can be illustrated by the below ERDs. The cardinality specifies that a student belongs to up to 1 department, while a department has multiple students.

Conversely, a department consists of multiple students. The nature of these relationships, whether one-to-one, one-to-many, or many-to-many, helps define the structure and behavior of the database. Understanding and properly defining relationships in an ERD is essential for accurately modeling the data and ensuring the integrity and efficiency of the database design.

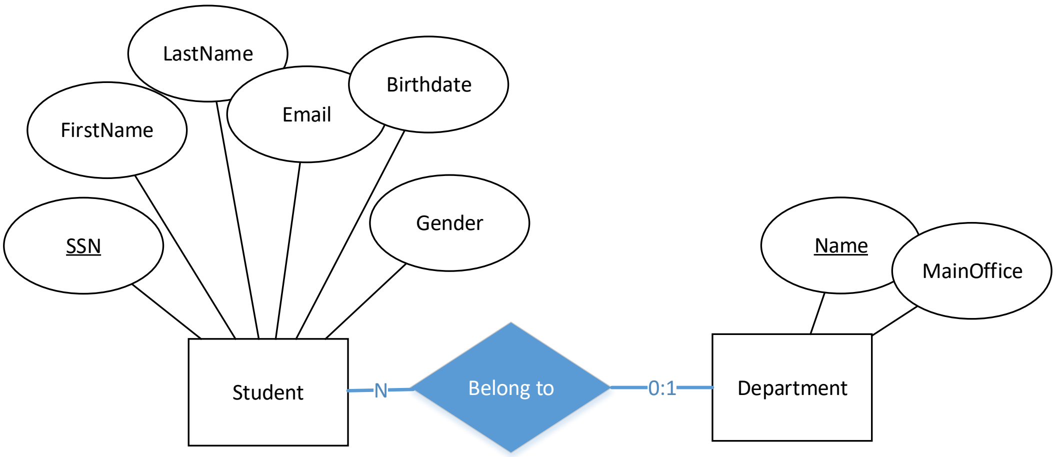


Figure 29. The relationship between Student and Department entity sets.

The relationship sometimes may have attributes to enrich the understanding of how entities are connected. For example, in the relationship *Has assessment result* between Subject and Student illustrated in Figure 30, attribute *Mark* presents the specific result while the attribute *Date* describes the date this results is recorded.

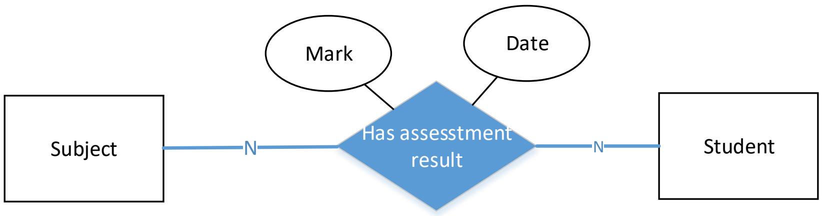


Figure 30. The Has assessment result *relationship consists of two attributes: Mark and Evaluation time.*

Sometimes, we may want to assert clearly that all instances of an entity set must take part in a relationship. For instance, a class is always an instance of a subject. In such cases, the double line is used to represent this **mandatory relationship,** as in Figure 31.

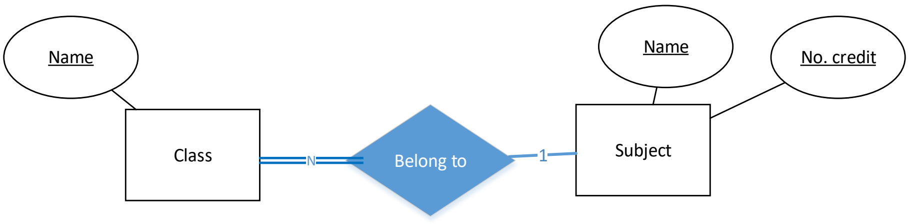


Figure 31. The Belong to relationship is mandatory for Class entity set.

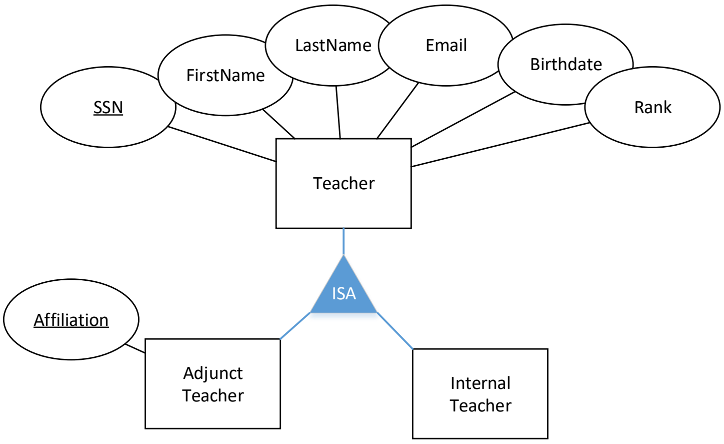


Figure 32. The Isa relationship between Teacher, Adjunct Teacher, and Internal Teacher entity sets.

A special case of relationship is the “**is-a**” which represents inheritance or specialization between entities. This type of relationship is also known as a subtype-supertype relationship. In an "is-a" relationship, a subtype entity inherits attributes and relationships from a supertype entity, indicating that the subtype is a more specialized version of the supertype. For example, as shown in Figure 32, in AMS, there might be a supertype entity called "Teacher," which contains common attributes such as SSN, FirstName, LastName, Email, Birthdate, Rank. Subtype entities such as "Adjunct Teacher" and "Internal Teacher" would then inherit these attributes from the "Teacher" entity but may also have additional attributes specific to their roles. The "is-a" relationship allows for the modeling of hierarchical structures and supports polymorphic behavior within the database schema. It helps in organizing entities into meaningful categories and capturing both shared and unique characteristics across different entity types.

c. *Weak Entity Set*

Sometimes, we cannot find any attribute set of an entity set to set as the key of its own. Instead, it relies on a related entity set, called the identifying or owner entity set, for its existence and identification. In such case, the dependent one is called a weak entity set. Weak entities often represent subordinate or dependent entities that cannot be uniquely identified without considering their relationship with another entity. To distinguish instances of a weak entity set, a combination of attributes from the weak entity set itself and the identifying entity set is used as a composite key. In AMS, the entity set Classroom is a weak entity set because each classroom instance cannot be uniquely identified without considering its association with the building it belongs to. In this case, the Building entity set serves as the identifying entity set, and the combination of the No. (the room number) and the Name of the building could form a composite key for the Classroom entity set.

In ERD, the rectangle representing a weak entity set has double lines to distinguish with the strong one (see Figure 33). The component attributes of the composite key is written with dotted underlines. The supporting relationship is drew is a double-line diamond.

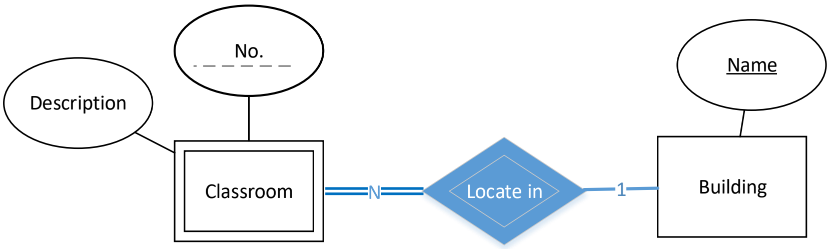


Figure 33. The Classroom is a week entity set, and the Locate in is its supporting relationship.

Weak entity sets are used in modeling real-world scenarios where entities depend on others for their identity and are crucial for maintaining data integrity and capturing complex relationships in the database schema.

d. Drawing ERD Guidelines

We have studied all the important concepts of ERDs. It is now the time to discuss how to come up with an ERD for a system**.** The database analyst must read the **functional requirement description** carefully to extract entity sets, attributes, and relationships by following these guidelines:

- Identify Nouns: Look for nouns or noun phrases that the system needs to manage in the functional requirement description. These often represent entities in the system. All relevant nouns of AMS have been highlighted with underlines in the section 3, such as “Academic year”, “name”, “starting date”, “ending date”.
- Define Entity Sets: Among the highlighted nouns, choose ones with the following criteria:
    - Nouns represent a distinct concept or object in the domain such as Subject, Student, and Teacher.
    - Object has more than one data element. The Department deserves to be an entity set because it has two attributes: Name and Main Office.
    - Nouns whose cardinality are many in a relationship such as Class (although class has only 1 real attribute Name), and AttendanceState.
    - A set of objects should exist as an entity set to support a weak entity set. For example, the Building is set as an entity set to identify classrooms even when it has only one attribute (Name). Someones can think about setting the Building Name and No. (Room number) as the key of the classroom to make the Classroom a strong entity set. However, this idea requires a constraint forcing the Building Name entered into the classroom must be valid while the list of buildings can be changed quickly.
- Determine Attributes: For each entity set, identify the attributes or properties that describe it. Attributes are typically adjectives or descriptors associated with entities.
- Consider Relationships: Analyze the interactions between different entities mentioned in the requirements. Look for verbs or verb phrases that describe actions or associations between entities. These indicate potential relationships. For instance, "places an order," "owns," or "purchases" imply relationships between entities like "customer" and "order," or "customer" and "product."
- Clarify Cardinality and Participation: Determine the cardinality (one-to-one, one-to-many, or many-to-many) and participation constraints (mandatory or optional) for each relationship based on the business requirements. For example, a customer may place many orders (one-to-many), but an order must belong to exactly one customer (mandatory participation).

- Review and Refine: Validate your choices by reviewing the requirements and ensuring that your entity sets, attributes, and relationships accurately reflect the domain and business needs. Refine your model as needed based on feedback and further analysis.
- Create an Entity-Relationship Diagram (ERD): Use your identified entity sets, attributes, and relationships to construct an ERD. The ERD visually represents the structure of the database, helping stakeholders understand the system's data model.

By following these guidelines, we can effectively translate business requirements into a structured and meaningful data model, laying the groundwork for successful database design and implementation. Figure 34 shows an example of ERD for AMS.

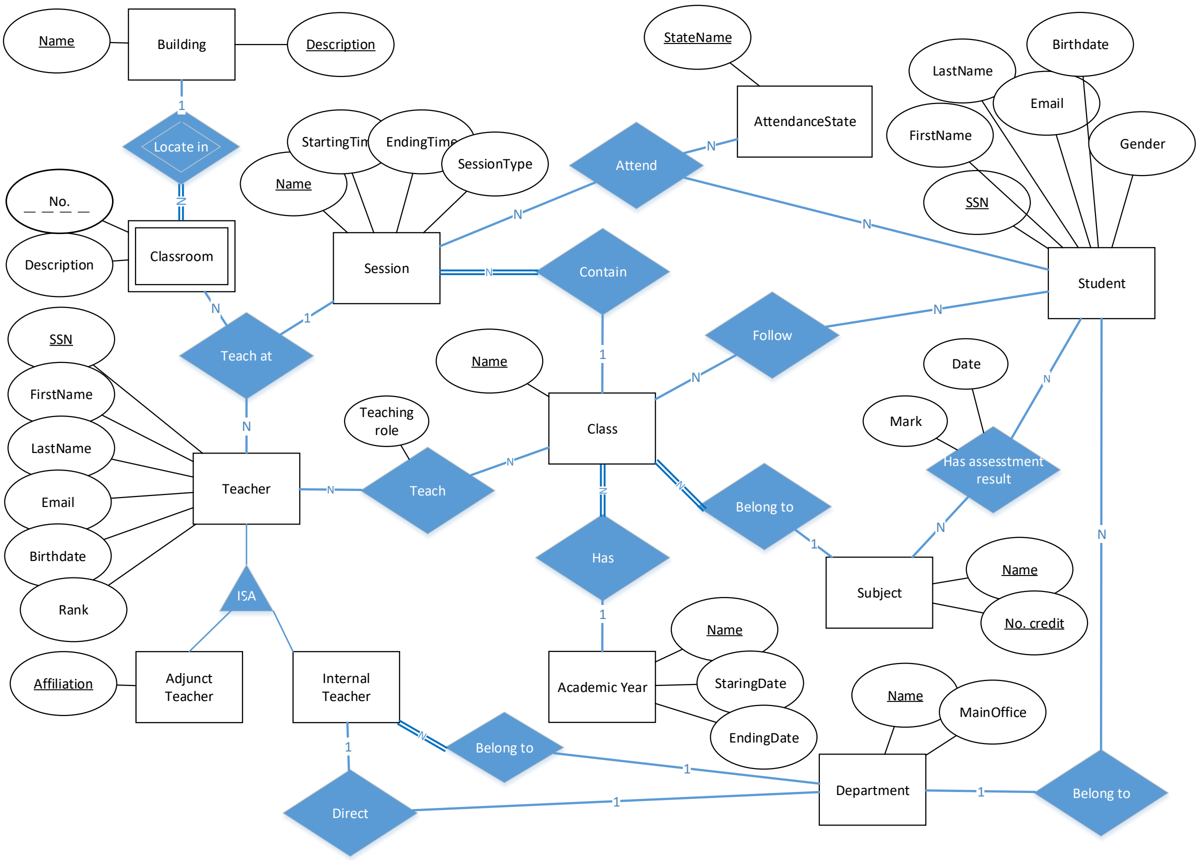


Figure 34. A sample ERD for AMS.

5. Database Design

a. Logical design

Once you have constructed an Entity-Relationship Diagram (ERD) representing the conceptual model of your database, the next step is to transform it into a logical database design which is independent from a specific RDBMS. This process involves converting the entities, attributes, and relationships identified in the ERD into database tables, columns, and constraints. Here's a step-by-step guide to transforming your ERD into a well-structured database design:

- Convert all entity set into tables. Each attribute of an entity set will become a column of the corresponding table with the specified data type and constraints. The key attribute will be the primary key of the table accordingly. For example, entity sets Department and Student result in tables Department (Name, MainOffice) and Student (SNN, FirstName, LastName, Email, BirthDate, Gender). Surrogate keys can be used to simpplied data of original primary keys.
- Process entity sets relating to the “is-a” relationship by one of three following approaches with the is-a relationship consisting Teacher, AdjunctTeacher, and IntenalTeacher:
  - Object-oriented style: create one table for each class in the hierarchy. The sample tables include Teacher (SSN, FirstName, LastName, Email, Birthdate, Rank) and AdjunctTeacher (SSN, Affiliation), IntertalTeacher (SSN, DepartmentID).
  - Use NULLs: create a table for all entity set in the hierarchy. As parent entity sets do not have attributes of their descendants, we need to fill NULL to the these column of the rows corresponding to the parent entity sets. The sample tables include Teacher (SSN, FirstName, LastName, Email, Birthdate, Rank, Affiliation, DepartmentID) where Affiliation, DepartmentID are nullable columns.
  - E/R style: define one table for each class at the leaf levels. The sample tables include AdjunctTeacher(SSN, FirstName, LastName, Email, Birthdate, Rank, Affiliation), InteralTeacher(SSN, FirstName, LastName, Email, Birthdate, Rank, DepartmentID).
- Handle the relationships in 3 cases of the cardinality:
  - Many-to-many: we create a new table having the primary key as the combination of key of particcipating entity set, and other columns corresponding to the attribute (if any) of the relationship. E.g. the table Teaching (TeacherID, ClassName, TeachingRole) corresponds to the relationship Teach between Teacher and Class.
  - One-to-many: add the primary key columns of the one-side table the many-side table to act as a foreign key. For example, since Student belong to Department, the table Student has a foreign key DepartmentName referring to the primary key Name of table Department.
  - One-to-one: add the primary columns of a table to other table as its foreign key. For instance, the direct relationship between Department and Intenal Teacher result in foreign key column DirectorSSN of table Department referring to the primary key SSN of table Teacher (SSN, FirstName, LastName, Email, Birthdate, Rank, Affiliation, DepartmentID).

- Normalize the Database: Analyze the tables and relationships to ensure the database schema adheres to normalization principles. Apply normalization techniques such as First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF) to eliminate data redundancy and maintain data integrity.
- Optimize Performance: Consider indexing strategies for frequently queried columns to improve database performance. Evaluate the use of views, stored procedures, and other database objects to optimize data retrieval and manipulation.
- Review and Iterate: Review the database design against the original requirements and the ERD to ensure alignment. Iterate on the design as necessary based on feedback, performance considerations, and evolving business needs.

By following these steps, we can effectively translate your ERD into a well-structured database design. The database schema of AMS is below:

- Department (ID, Name, MainOffice, DirectorID)
- Teacher (ID, FirstName, LastName, SSN, BirthDate, Email, Rank, Affiliation, DepartmentID)
- Student(ID, FirstName, LastName, Birthdate, Email, Gender, DepartmentID)
- AcademicYear (ID, Name, StartingDate, EndingDate)
- Subject (ID, Name, NoCredits)
- Class (ID, Name, SubjectID, AcademicYearID)
- ClassStudent (StudentID, ClassID)
- TeachingRole (ID, RoleName)
- ClassTeacher (ClassID, TeacherID, TeachingRoleID)
- Building (Name, Description)
- Classroom (ID, No., BuildingName, Description)
- Session (ID, Name, ClassID, ClassroomID, StaringTime, EndingTime, SessionType)
- AttendanceState (ID, StateName)
- Attendance (StudentID, SessionID, AttendanceStateID)
- Assesement (StudentID, SubjectID, Date, Mark)

The database designer should check the normal forms of obtained tables. The database mentioned above is already sufficiently normalized. See section “c. Normal forms” for more details about database normalization.

The data constraints need also be defined at this step. For AMS database, beside the keys, there are some below constraints:

- Teacher table: if the DepartmentID is NULL, the Affiliation must be defined, implying that this is a conjunction teacher.

- Student table: Gender must be one of three values: 1 (male), 2 (female), or 3 (other).
- Assessment table: according to the rule of this univesity, the value of Mark must be from 0 to 20.
- AcademicYear: the StartingDate must be before the EndingDate
- Session: the StartingTime must be before the EndingTime, and the SesssionType is one of (“Lecture”, “Practice”, “Tutorial”).

b. Physical Design

The physical design is obtained by translating the logical design into an actual database in a specific RDMBS. Some following aspects need to be consider to improve the quality of physical design by using techniques and features of the chosen RDMBS [6].

- Data constraints and integrity: define appropriate primary keys, foreign keys, unique constraints, check constraints, not null constraints, triggers.
- Performance: we first identify the possible heavy SQL statements, then consider performance optimization strategies such as defining indexes for searched or joined columns, use stored procedures and functions, use horizontal (row-based) or vertical (column-based) partitioning to divide large tables into smaller, more manageable pieces. For performance gains, denormalization can reduce joining operations for read-heavy workloads. With big databases, the storage need also be considered to avoid data overflow and to ensure the performance
- Convenience and maintainability: define views to simplify and clarify data, stored procedure and functions to provide reusable programming modules.

Getting back to the case study of AMS, its database schema in MySQL is depicted in Figure 35. There should be additional objects as follows:

- Check constraints and triggers are used to ensure data constraints discussed in the previous section.
- The indexes for FirstName, LastName, and Email of Students as they are included in high-frequency queries, and the Student table may have hundred thousand of records.
- Store procedures to query the student list of a class, the list of student of a class with the information who are eligible to take the test, a student's assessment result, a class's timetable, the total teaching hours of teachers in an academic year, and the top n (or n percent) students with the highest GPA in each academic year….

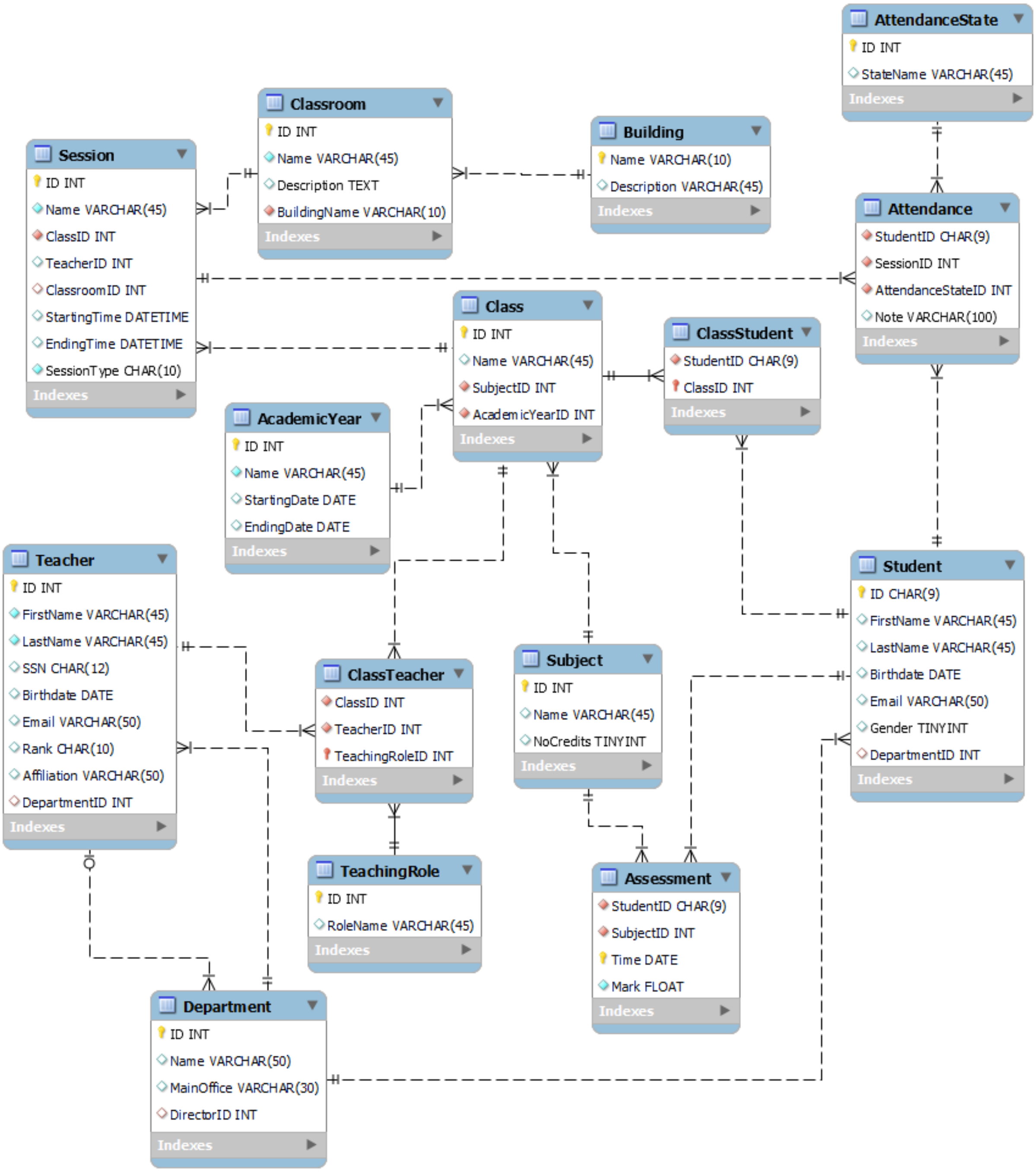


Figure 35. A sample database design for AMS.

c. *Normal forms*

In database design, normalization is a systematic approach to organizing data to minimize redundancy and ensure data integrity. Normal forms are a series of guidelines for determining the optimal table structure in a relational database. Each normal form addresses specific types of anomalies and issues that can arise from improper data organization. The process typically starts from the First Normal Form (1NF) and progresses through higher normal forms, each adding more constraints and structure.

First Normal Form (1NF)

A table is in the First Normal Form (1NF) if it meets the following criteria:

- **Atomicity:** Each column must contain atomic (indivisible) values. Multi-valued attributes are not allowed.
- **Uniqueness:** Each row must be unique, distinguishable by a primary key.
- **Consistent Data Types:** Each column must contain values of a single data type.

Consider the table AssesmentResult, which stores students' assessment results on subjects in Figure 36. This table is not in 1NF because the Dates and Marks columns contains multiple values.

AssesmentResult

StudentID	SubjectID	Dates	SubjectName	No.Credit	Marks
BI12-001	1	2024-11-20, 2024-12-15	Philosophy	3	9, 8, 13
BI12-002	2	2024-11-20	Basic Programming	4	12

Figure 36. AssesmentResultstable is not in 1NF.

To convert it to 1NF, we must ensure each column holds atomic values by split multi-values cells into data in different rows as in Figure 37:

AssesmentResult

StudentID	SubjectID	Date	SubjectName	No.Credit	Mark
BI12-001	1	2024-11-20	Philosophy	3	9
BI12-001	1	2024-12-5	Philosophy	3	8
BI12-001	1	2025-01-15	Philosophy	3	13
BI12-003	2	2024-11-20	Basic Programming	4	12

Figure 37. AssesmentResult is normalized in 1NF.

Second Normal Form (2NF)

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

- It is in 1NF.
- All non-key attributes are fully functionally dependent on the primary key. This means that each non-key attribute must depend on the entire primary key, not just part of it.

The table AssesmentResults in Figure 37 has (StudentID, SubjectID, Date) as the primary key; however, (SubjectID) → (SubjectName, No.Credit) so it does not satisfy 2NF. To fix this, we decompose the table into 2 new tables AssesmentResults and Subjects as in Figure 38.

AssesmentResults

StudentID	SubjectID	Date	Mark
BI12-001	1	2024-11-20	9
BI12-001	1	2024-12-5	8
BI12-001	1	2025-01-15	13
BI12-003	2	2024-11-20	12

Subjects

SubjectID	SubjectName	No.Credit
1	Philosophy	3
2	Basic Programming	4

Figure 38. AssesmentResults and Subjects are in 2NF.

Third Normal Firm (3NF)

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

- It is in 2NF.
- There are no transitive dependencies, where non-key attributes depend on other non-key attributes. In 3NF, every non-key attribute must depend only on the primary key.

The table Student in Figure 39, the functional dependencies (StudentID) → (DeptName) is a transitive dependencies as (DeptID) → (DeptName). This violates 3NF.

Student

StudentID	FullName	BirthDate	DeptID	DeptName
BI12-001	Nguyen Ngoc Ky	2003-10-20	1	General Education Department
BI12-001	Nguyen Son Lam	2003-12-20	13	Foreign Language Center

Figure 39. Student is not in 3NF.

To obtain 3NF, we decompose the table into 2 tables Student and Department as in Figure 40.

Student

StudentID	FullName	BirthDate	DeptID
BI12-001	Nguyen Ngoc Ky	2003-10-20	1
BI12-001	Nguyen Son Lam	2003-12-20	13

Department

DeptID	DeptName
1	General Education Department
13	Foreign Language Center

Figure 40. Student and Department are in 3NF.

Higher Normal Forms

There are higher normal forms beyond 3NF, such as the Boyce-Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF). However, they are rarely considered in practice. Each of these addresses more specific types of anomalies and dependencies:

- **BCNF:** A table is in BCNF if it is in 3NF and for every functional dependency (A → B), A is a superkey.
- **4NF:** A table is in 4NF if it is in BCNF and has no multi-valued dependencies.
- **5NF:** A table is in 5NF if it is in 4NF and every join dependency in the table is implied by the candidate keys.

Highlight & Summary

In this chapter, we embark on a journey through the intricate world of database analysis and design. From conceptualizing data models to implementing efficient database schemas, this chapter explores the fundamental principles and methodologies essential for creating robust, scalable, and maintainable databases. The real-world example of AMS provide readers with hands-on experience in database analysis and design.

Exercises

1. What is an ERD, and what is its purpose in database design?
2. Describe the key components of an ERD, including entities, attributes, relationships, and cardinality.
3. Explain the difference between an entity and an attribute in an ERD. Provide examples.
4. What are the cardinality constraints in an ERD, and how do they represent the relationship between entities?
5. How do you represent different types of relationships (e.g., one-to-one, one-to-many, many-to-many) in an ERD?
6. Can an entity have more than one relationship with another entity? If so, how is this represented in an ERD?
7. Describe the process of converting an ERD into a relational database schema. What are the steps involved?
8. Why is normalization important in database design?
9. What are the requirements for a table to be in 1NF, 2NF, 3NF?
10. Explain the trade-offs between normalization and denormalization in database performance and maintenance.

References

[1] Jeffrey D. Ullman and Jennifer Widom. 2014. First Course in Database Systems (3rd. ed.). Prentice Hall Press, USA

[2] E. F. Codd. 1970. A relational model of data for large shared data banks. Commun. ACM 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[3] Neeraj Sharma, Liviu Perniu, Raul F. Chong, Abhishek Iyer, Adi-Cristina Mitea, Chaitali Nandan, Mallarswami Nonvinkere, Mirela Danubianu. 2010. Database Fundamentals. IBM Corporation.

[4] Rahul Batra. 2018. SQL Primer: An Accelerated Introduction to SQL Basics (1st. ed.). Apress, USA.

[5] Peter Pin-Shan Chen. 1976. The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. 1, 1 (March 1976), 9–36. https://doi.org/10.1145/320434.320440

[6] S. S. Lightstone, T. Teorey, and T. Nadeau, Physical Database Design, Morgan-Kaufmann, San Francisco, 2007

Appendix: Report structure for a mini project with database

This section provide a sample structure for a report on the development of a relational database project, encompassing various phases from initial context and background research to final SQL implementation and testing. Students can use this structure as a reference to write the report for the final project at the end of the course.

Table of Contents

- List all sections and subsections with page numbers.

Abstract

- A brief summary of the project, including objectives, methodology, and key findings.

1. Introduction

a. Context

- Describe the context in which the project is being undertaken.
- Explain the problem or need that the database project addresses.

b. Background

- Provide background information, concepts relevant to the project.
- Discuss previous work or existing systems related to the project.

c. System Definition

- Define the goals
- Provide the scope the project.
    - Boundaries and limitations of the project.
    - State the objectives and deliverables.

2. Introduction

a. User Requirements by Use Cases

- Include use case diagrams and descriptions to describe functional requirements.
- Present use case scenarios to illustrate how users will interact with the database.

b. Data Analysis

- From the description of use cases, explain the data (type of objects and their properties, relationship, constraints …) the system will manage.
- Present the Entity-Relationship Diagram (ERD) to illustrate the conceptual schema.
- Discuss the entities, attributes, and relationships depicted in the ERD.

3. Design

a. Logical Design

- Translate the ERD into a conceptual design which
- Define data constraints.
- Define entities, attributes, and relationships in more detail.
- Normalize the tables if necessary.

b. Physical Design

- Describe the physical implementation of the database in a specific RDMBS.
- Discuss indexing, storage considerations, programmable objects (procedures, functions, triggers), and performance optimization strategies.

4. SQL Implementation

a. Database Creation

- Provide SQL scripts used to create the database schema.
- Include CREATE TABLE statements and any constraints (primary keys, foreign keys, etc.).

b. Data Insertion

- Provide SQL scripts or procedures for inserting initial data into the database.
- Include INSERT statements with sample data.

c. Queries and Reports

- Present SQL statements corresponding to the functions defined at section 2.a:
    - How to perform data manipulation tasks (insert, update, delete data),
    - How to retrieve data from the database (for displaying on reports, on GUI).
- Discuss any views, stored procedures, or triggers implemented.

d. Testing and Validation

- Describe the testing process used to validate the database design and implementation.
- Provide examples of test cases and their results.
- Discuss any issues found and how they were resolved.

5. Conclusion

- Summarize the key findings and results of the project.
- Discuss the significance of the project and its impact.
- Suggest future work or improvements that could be made.

6. References

- List all sources and references used in the report, formatted according to a specified citation style.

7. Appendices

- Include any additional material that supports the report but is too detailed to include in the main sections.
- Examples: detailed data models, complete SQL scripts, additional diagrams.