Lecture Notes in Database Management Systems

Assoc. Prof. Mohammed Abd Elsalam Ahmed
Information Systems Department
Faculty of Commerce & Business Administration
Helwan University

Dr. Mennah Ibrahim Gabr
Information Systems Department
Faculty of Commerce & Business Administration
Helwan University

الناشر: جهاز نشر وتوزيع الكتاب
الجامعي- جامعة حلوان

الفصل الدراسى الأول

2023-2024

This material is specially made for the undergraduate students. It contains what will be explained in the lectures. The selected topics have been collected from various books, websites, and open sources, that have been edited and summarized. Its only purpose is to help the students to follow what will be explained in the lectures and present it in an easy and suitable way.

This Material is prepared and collected to help students at Level 3 – Business Information systems Program - faculty of commerce – Helwan university.

Table of content

Chapter 1

Introduction to DBMS

1.1 introduction

All organizations public, governmental or private, small or large depend on computerized information systems for carrying out their daily activity. At the heart of each such information system, there is a database. At a very general level, we can define a database as a persistent collection of related data, where data are facts that have an implicit meaning. For instance, an employee's name, social security number, or date of birth are all facts that can be recorded in a database. Typically, a database is built to store logically interrelated data representing some aspects of the real world, which must be collected, processed, and made accessible to a given user population. The database is constructed according to a data model which defines the way in which data and interrelationships between them can be represented. The collection of software programs that provide the functionalities for defining, maintaining, and accessing data stored in a database is called a database management system (DBMS).

A Database-Management System (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the database, contains information relevant to an enterprise.

The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and techniques form the focus of this book. This chapter briefly introduces the principles of database systems.

1.2 Purpose of Database Systems:

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data,

keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- Add new students, instructors, and courses
- Register students for courses and generate class rosters
- Assign grades to students, compute grade point averages (GPA), and generate transcripts.

System programmers wrote these application programs to meet the needs of the university. New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major (say, computer science). As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, etc. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. Thus, as time goes by, the system acquires more files and more application programs.

This typical file-processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) were introduced, organizations usually stored in such systems.

1.3 Disadvantages of keeping organizational information in a file-processing system:

- **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the previous files are likely to have different structures and the programs may be written in several programming language. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency;** that is, the various copies of the same data may no longer

agree. For example, a changed student address may be reflected in the music department records but not elsewhere in the system.

- **Difficulty in accessing data**. Suppose that one of the university clerks asks the data-processing department to generate such a list. There is, however, an application program to generate the list of *all* students. The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist, again, the clerk has the preceding two options, neither of which is satisfactory. The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient manner. More responsive data-retrieval systems are required for general use.
- **Data isolation**. Because data are scattered in various files, and files may be in different formats,

writing new application programs to retrieve the appropriate data is difficult.

- **Integrity problems**. The data values stored in the database must satisfy certain types of *consistency constraints*. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.
- **Atomicity problems**. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $500 was removed from the balance of execution of the program, it is possible that the $500 was removed from the balance of department A, but was credited to the balance of department *B*, resulting in an inconsistent database state. Clearly,

it is essential to database consistency that either both the credit and debit occur, or that neither occur. It is difficult to ensure atomicity in a conventional file-processing system.

- **Concurrent-access anomalies**. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000 and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department A

may contain $9500 or $9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously. As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

- **Security problems**. Not every user of the database system should be able to access all the data. For example, in a university, payroll personal need to

see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult. These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a university organization as a running example of a typical data-processing application.

1.4 Characteristics and Benefits of a Database

There are several characteristics that distinguish the database approach from the file-based system or approach. This chapter describes the benefits (and features) of the database system.

Self-describing nature of a database system

A database system is referred to as *self-describing* because it not only contains the database itself, but also metadata which defines and describes the data and relationships between tables in the database.

This information is used by the DBMS software or database users if needed. This separation of data and information about the data makes a database system totally different from the traditional file-based system in which the data definition is part of the application programs.

Insulation between program and data

In the file-based system, the structure of the data files is defined in the application programs so if a user wants to change the structure of a file, all the programs that access that file might need to be changed as well.

On the other hand, in the database approach, the data structure is stored in the system catalogue and not in the programs. Therefore, one change is all that is needed to change the structure of a file. This insulation between the programs and data is also called program-data independence.

Support for multiple views of data

A database supports multiple views of data. A *view* is a subset of the database, which is defined and dedicated for particular users of the system. Multiple users in the system might have different views of the system. Each

view might contain only the data of interest to a user or group of users.

Sharing of data and multiuser system

Current database systems are designed for multiple users. That is, they allow many users to access the same database at the same time. This access is achieved through features called *concurrency control strategies*. These strategies ensure that the data accessed are always correct and that data integrity is maintained.

The design of modern multiuser database systems is a great improvement from those in the past which restricted usage to one person at a time.

Control of data redundancy

In the database approach, ideally, each data item is stored in only one place in the database. In some cases, data redundancy still exists to improve system performance, but such redundancy is controlled by application programming and kept to minimum by introducing as little redundancy as possible when designing the database.

Data sharing

The integration of all the data, for an organization, within a database system has many advantages. First, it allows for data sharing among employees and others who have access to the system. Second, it gives users the ability to generate more information from a given amount of data than would be possible without the integration.

Enforcement of integrity constraints

Database management systems must provide the ability to define and enforce certain constraints to ensure that users enter valid information and maintain data integrity. A *database constraint* is a restriction or rule that dictates what can be entered or edited in a table such as a postal code using a certain format or adding a valid city in the City field.

There are many types of database constraints. *Data type*, for example, determines the sort of data permitted in a field, for example numbers only. *Data uniqueness* such as the primary key ensures that no duplicates are entered. Constraints can be simple (field based) or complex (programming).

Restriction of unauthorized access

Not all users of a database system will have the same accessing privileges. For example, one user might have *read-only access* (i.e., the ability to read a file but not make changes), while another might have *read and write privileges*, which is the ability to both read and modify a file. For this reason, a database management system should provide a security subsystem to create and control different types of user accounts and restrict unauthorized access.

Data independence

Another advantage of a database management system is how it allows for data independence. In other words, the system data descriptions or data describing data (metadata) are separated from the application programs. This is possible because changes to the data structure are handled by the database management system and are not embedded in the program itself.

Transaction processing

A database management system must include concurrency control subsystems. This feature ensures that data remains consistent and valid during transaction

processing even if several users update the same information.

Provision for multiple views of data

By its very nature, a DBMS permits many users to have access to its database either individually or simultaneously. It is not important for users to be aware of how and where the data they access is stored.

1.5 Database-System Applications

Databases are widely used. Here are some representative applications:

- **Enterprise information**
  - **Sales:** For customer, product, and purchase information.
  - **Accounting:** For payments, receipts, account balances, assets and other accounting information.
  - **Human resources:** For information about employees, salaries, payroll taxes, and benefits and for generation of paychecks.
  - **Manufacturing:** For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.

- **Online retailers:** For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

- **Banking and finance:**
  - **Banking:** For customer information, accounts loans, and banking transactions.
  - **Credit card transactions:** For purchases on credit cards and generation of monthly statements.
  - **Finance:** For storing information about holdings, sales and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
  - **Universities:** For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
  - **Airlines:** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
  - **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards,

and storing information about the communication networks.

As the list illustrates, databases form an essential part of every enterprise today, storing not only types of information that are common to most enterprises, but also information that is specific to the category of the enterprise. Over the course of the last four decades of the twentieth century, use of databases grew in all enterprises. In the early days, very few people interacted directly with database systems, although without realizing it, they interacted with databases indirectly– through printed reports such as credit card statements, or through agents such as bank tellers and airline reservation agents. Then automated teller machines came along and let users interact directly with databases. Phone interfaces to computers (interactive voice-response systems) also allowed users to deal directly with databases – a caller could dial a number, and press phone keys to enter information or to select alternative options, to find flight arrival/departure times, or to register for courses in a university.

The Internet revolution of the late 1990s sharply increased access to databases. Organizations converted many of their phone interfaces to databases into web interfaces and made a variety of services and information

available online. For instances, when are accessing data stored in a database. When you enter an order online, your order is stored in a database. When you access a bank web site and retrieve your bank balance and transaction information, the information is retrieved from the bank's database system. When you access a web site, information about you may be retrieved from a database to select which advertisements you should see. Furthermore, data about your Web accesses may be stored in a database.

Thus, although user interfaces hide details of access to a database, and most people are not even aware they are dealing with a database, accessing databases forms an essential part of almost everyone's life today.

The importance of the database systems can be judged in another way- today, database system vendors like Oracle are among the largest software companies in the world, and database systems form an important part of the product line of Microsoft and IBM.

1.6 Data Abstraction:

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database system users are not

computer trained, developers hide the complexity from users through several levels abstraction, to simplify user's interactions with the system:

- **Physical level**. The lowest level of abstraction describes *how* the data are actually stored the physical level describes complex low-level data structures in detail.
- **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
- **View level**. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of

abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Figure 1.1 shows the relationships among the three levels of abstraction. An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction.

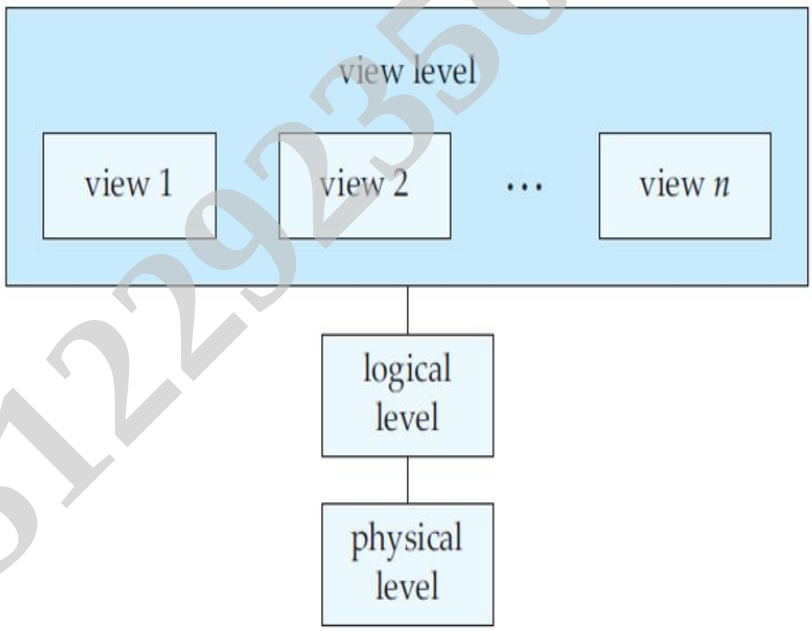


Figure 1.1 The three levels of data abstraction.

Many high-level programming languages support the notion of a structured type. For example, we may describe a record as follows:

type *instructor* = **record**
ID : **char** (5);
Name : **char** (20);
dept name : **char** (20);
Salary : **numeric** (8,2);
end;

This code defines a new record type called instructor with four fields. Each field has a name and a type associated with it. A university organization may have several record types, including.

- *department,* with fields *dept name*, *building*, and *budget*
- *student,* with fields *ID*, *name*, *dept name*, and *tot cred*

At the physical level, an instructor, department, or student record can be described as a block of consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest- level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well.

Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction. Finally, at the view level, computer users see s set of application programs that hide details of data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database.

For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

1.7 Instances and schemas:

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. Database schema corresponds to the variable declarations (along with associated type

definitions) in a program. Each variable has a particular value at a given instant. The values of the variables bin a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the level of abstraction. **The physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas, sometimes called **subschemas** that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

1.8 Data Models Categories

The data models can be classified into four different categories:

- **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.
- **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.
- **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an

object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

- **Semi structured Data Model**. The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. **The Extensible Markup Language (XML)** is widely used to represent semi structured data.

1.9 Database Design

Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation. They are part of the operation of some enterprise whose end product may be information from the database or may be some device for which the database plays only a supporting role.

Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader

set of issues. In this text, we focus initially on the writing of database queries and the design of database schemas.

1.10 Data Storage and Querying

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is approximately 1000 megabytes (actually 1024) (1 billion bytes), and a terabyte is 1 million megabytes (1 trillion bytes). Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system to simplify and facilitate access to data. The query processor allows database users to obtain good

performance while being able to work at the view level and not be burdened with understanding the physical-level details of the implementation of the system. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

1.11 Transaction Management

Often, several operations on the database form a single logical unit of work. An example is a funds transfer, one department account (say *A*) is debited and another department account (say *B*) is credited. Clearly, it is essential that either both the credit and debit hour, or that neither occur. That is, the funds transfer must happen in its entirety or not at all. This all-or-none requirement is called **consistency**. Finally, after the successful execution of a funds transfer, the new values of the balances of accounts *A* and *B* must persist, despite the possibility of system failure. This persistence requirement is called **durability**.

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any

database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of *A* or the credit of *B* must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs. It is the programmer's responsibility to define properly the various transactions so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account of department *A* to the account of department *B* could be defined to be composed of two separate programs: one that debits account *A*, and another that credits account *B*. The execution of these two programs by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions. Ensuring the atomicity and durability properties is the responsibility of the database system-specifically, of the recovery manger. In the absence of failures, all transactions complete successfully, and atomicity is achieved easily. However, because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be

restored to the state in which it was before the transaction in question started executing. The database system must therefore perform failure recovery, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the concurrency-control manger to control the interaction among the concurrent transactions to ensure the consistency of the database. The transaction manager consists of the concurrency-control manager and the recovery manager.

1.12 Database users

In the context of databases, users play a crucial role in accessing and managing data. Here's a detailed overview of database users and their roles. However, lets first know what is meant by Database User?

A ***database user*** refers to an individual or an application that interacts with a database system. Users are granted specific privileges and permissions to perform operations on the database. They can be categorized into different types based on their roles and responsibilities.

It's important to note that the specific roles and responsibilities of users may vary depending on the organization, database management system, and the specific requirements of the database environment.

1. ***Database Administrator (DBA):***

   The Database Administrator is responsible for the overall management and maintenance of the database system. They handle tasks such as database installation, configuration, security, performance tuning, backup and recovery, and user management. DBAs have elevated privileges to perform administrative tasks.

2. ***Application Developers:***

   Application developers are users who design and develop software applications that interact with the database. They create database schemas, write queries, and implement the business logic required for the applications. Developers require appropriate access permissions to perform their tasks effectively.

3. ***Data Analysts:***

   Data analysts are users who extract, analyze, and interpret data from the database to derive meaningful insights. They often work with business intelligence tools and reporting systems to generate reports, visualize data, and identify trends or patterns. Data analysts require read access to relevant database tables and views.

4. ***Data Scientists:***

   Data scientists are users who employ advanced statistical and analytical techniques to extract knowledge and make predictions from data. They utilize machine learning algorithms, statistical models, and data mining techniques to uncover hidden patterns and valuable insights. Data scientists typically need read and write access to perform their analyses.

5. ***Data Entry Operators:***

   Data entry operators are responsible for entering data into the database system. They input data from various sources, ensuring accuracy and consistency. Data entry operators require appropriate access permissions and may have restricted privileges to perform their tasks.

6. ***Business Users:***

   Business users are individuals who use the database system to access and retrieve data for their day-to-day operations. They may include managers, executives, sales representatives, or customer support personnel. Business users require read access to relevant data to perform their job functions efficiently.

7. ***External Users:***

   External users are individuals or entities who are granted limited access to the database system from outside the organization. Examples include customers, clients, vendors, or partners who may require access to specific data or services. External users have restricted privileges and access rights.

Following are other Important Points regarding database roles:

- User Roles and Permissions: Database systems often implement role-based access control (RBAC), where users are assigned specific roles with predefined permissions. Roles group related privileges together, simplifying user management and ensuring security.

- Authentication and Authorization: Database systems employ authentication mechanisms to verify the identity of users before granting access. Authorization mechanisms control user access based on their roles and permissions, ensuring data security and integrity.

- User Privileges: Privileges determine the operations a user can perform on the database objects. They include read, write, update, delete, create, and execute privileges. Privileges can be granted at various levels, such as database level, table level, or column level.
- User Management: Database administrators are responsible for creating, modifying, and deleting user accounts. They also handle password management, account locking, and password expiration policies to ensure secure user access.

- Auditing and Logging: Database systems often provide auditing and logging features to track user activities. This helps monitor user actions, detect unauthorized access attempts, and maintain an audit trail for compliance and security purposes.

Chapter 2
Relational Model

2.1 Introduction

The relational model is today primary data model for commercial data processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. In this chapter, we first study the fundamentals of the relational model. A substantial theory exists for relational database.

In this chapter, we concentrate on describing the basic principles of the relational model of data. We begin by defining the modeling concepts and notation of the relational model in Section 2.2. Section 2.3 is devoted to a discussion of relational constraints that are considered an important part of the relational model and are automatically enforced in most relational DBMSs. Section 3.4 defines the update operations of the relational model, discusses how violations of integrity constraints are handled, and introduces the concept of a transaction.

2.2 Relational Model Concept

A relational database consists of a collection of tables, each of which is assigned a unique name. For example, consider the instructor table of table 2.1, which stores information about instructors. The table has four column

headers: ID, name, dept name, and salary. Each row of this table records information about an instructor, consisting of instructor's ID, name, dept_name, and salary.

Table 2.1 the instructor relation

ID	Name	Dept_name	Salary
10101	Mohamed	IT	6500
12121	Ashraf	Finance	10000
15151	Ayman	Accounting	15000
22222	Shady	Finance	12000
32343	Mai	Management	5000
33456	Iman	Biology	18000
45565	Emad	IT	16000
58583	Sayed	Finance	16000
76543	Sally	Biology	11000

Similarly, the course table of table 2.2 stores information about courses, consisting of a course id, title, dept_name, and credits, for each course. Note that each instructor is identified by the value of the column ID, while each course is identified by the value of column course id.

Table 2.2 the course relation

Course_id	Title	Dept_name	credits
BIO-101	Intro. To Biology	Biology	4
BIO-399	Genetics	Biology	4
IT-102	Intro to computer	IT	2
IT-201	Advanced DB	IT	3
IT-365	Image processing	IT	4
FIN-258	Managerial financing	Finance	3
FIN-109	Investment Banking	Finance	4
ACC-601	Corporation	Accounting	3
ACC-511	Auditing	Accounting	3

In general, a row in a table represents the relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name. In a mathematical terminology, a tuple is simply a sequence or list of values. A relationship between n values is represented mathematically by an n-tuple of values, i.e., a tuple with n values, which corresponds to a row in a table.

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table. Each attribute has domain. A **domain** D is a set of atomic values. By atomic we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn.

Examining table 2.1, we can see that the relation instructor has four attributes: ID, name, dept_name, and salary. We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of instructor shown in table 2.1 has 9 tuples, corresponding to 9 instructors. The domain of credits attribute is (2, 3, or 4). In this chapter, we shall be using several different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. They do not include all the data an actual university database would contain, to simplify our presentation.

2.3 Relational Database Schema

When we talk about a bout database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time.

The concept of relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition. In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time; similarly, the content of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change. Although it is important to know the difference between a relation schema and a relation instance. Where required, we explicitly refer to the schema or to instance, for example “the instructor schema,” or “an instance of the instructor relation.” However, where it is clear whether we mean the schema or the instance, we simply use the relation’s name.

Note that the attribute dept_name appears in both the instructor schema and the department schema. This duplication is not a coincidence. Rather, using common attributes in relation schema is one way of relating tuples of distinct relations.

For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the department relation to find the dept_name of all the departments housed in Watson. Then, for each such department, we look in the instructor relation to find the information about the instructor associated with the corresponding dept_name.

Table 2.3 department relation

Dept_name	building	budget
Biology	Watson	90000
IT	Taylor	100000
Finance	Painter	120000
Accounting	Taylor	80000

2.4 Relational Model Constraints

In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into four categories. Which include domain

constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

2.4.1 Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain (A). The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types. Other possible domains may be described by a sub range of values from a data type or as an enumerated data type in which all possible values are explicitly listed.

2.4.2 Key Constraints

All tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. Usually, there are other subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes.

A super key is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a super key. The name of attribute of *instructor*, on the other hand, is not a super key, because several instructors might have the same name. Formally, let *R* denote the set of attributes in the schema of relation *r*. if we say that a subset *K* of *R* is a **super key** for *r*, we are restricting consideration of instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. that is, if *t1* and t2 are in r and *t1* = *t2*, then *t1.K* = *t2.K*.

A **super key** may contain extraneous attributes. For example, the combination of *ID* and *name* is a super key for the relation *instructor*, if *K* is a super key, then so is any superset of *K*. We are often interested in super keys for which no proper subset is a super key. Such minimal super keys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both {*ID*} and {*name, dept name*} are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, {*ID, name*}, does not form a

candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled. Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name.

In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social-security numbers, international enterprises must generate their own unique identifiers. An alternative is to use some unique combination of other attributes as a key. The primary key should be chosen such that its attributes values are never, or very rarely, changed. For instance, the *address* field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, expect

if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique. It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

2.4.3 Constraints on NULL Values

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, in Table 2.4, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone does not apply to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone, or he has one but we do not know it (value is unknown).

Table 2.4: The student relation

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

2.4.4 Entity Integrity and Referential Integrity Constraints

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another

relation must refer to an existing tuple in that relation. For example, in table 2.5, the attribute Dnum of Project gives the department number for which each project; hence, its value in every project tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

Table 2.5 Department and Project Relations

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

To define referential integrity more formally, first we define the concept of a foreign key. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R1 and R2. A set of attributes FK in relation schema R1 is a **foreign key** of R1 that references relation R2 if it satisfies the following rules:

- The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.

- A value of FK in a tuple t1 of the current state r1 (R1) either occurs as a value of PK for some tuple t2 in the current state r2 (R2) or is NULL. In the former case, we have t1 [FK] = t2 [PK], and we say that the tuple t1 references or refers to the tuple t2.

In this definition, R1 is called the referencing relation and R2 is the referenced relation. If these two conditions hold, a referential integrity constraint from R1 to R2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

2.5 Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by schema diagrams.

Figure 2.1 shows the schema diagram for Company. Each relation appears as a row, with the relation's name at the top in blue, and the attributes listed inside the row. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation. Referential integrity constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram later. Many database systems provide design tools with a graphical user interface for creating diagrams.

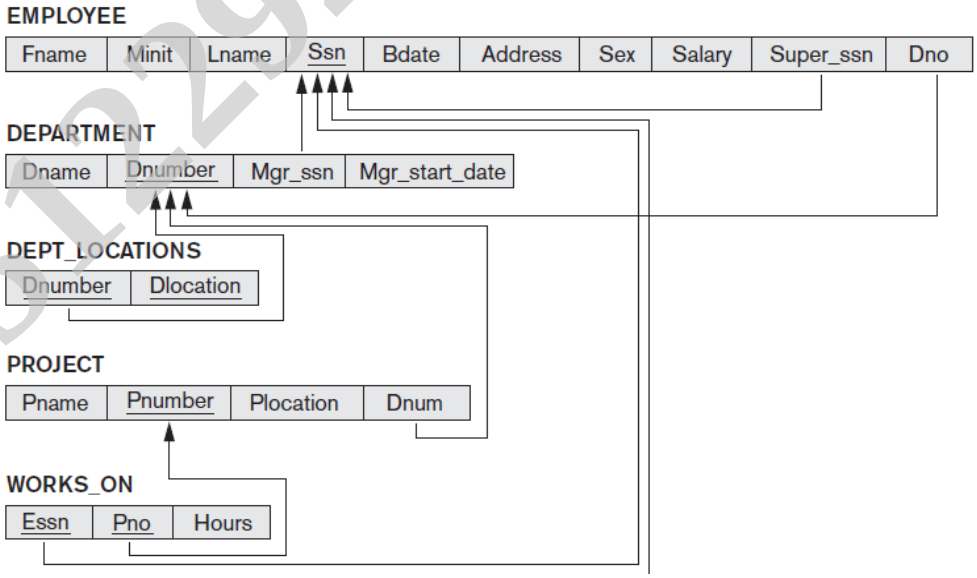


Figure 2.1: Referential integrity constraints displayed on the company relational database schema.

Chapter 3

Entity Relationship Modeling

3.1 Introduction

Conceptual modeling is a very important phase in designing a successful database application. Generally, the term database application refers to a particular database and the associated programs that implement the database queries and updates. For example, a BANK database application that keeps track of customer accounts would include programs that implement database updates corresponding to customer deposits and withdrawals. These programs provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application— the bank tellers, in this example. Hence, a major part of the database application will require the design, implementation, and testing of these application programs. Traditionally, the design and testing of application programs has been considered to be part of software engineering rather than database design. In many software design tools, the database design methodologies and software engineering methodologies are intertwined since these activities are strongly related.

In this chapter, we follow the traditional approach of concentrating on the database structures and constraints during conceptual database design. The design of application programs is typically covered in software

engineering courses. We present the modeling concepts of the Entity-Relationship (ER) model, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications. We also present the diagrammatic notation associated with the ER model, known as ER diagrams.

In this chapter, we introduced data models. A data model is an integrated collection of concepts that represents real world objects, events, and their relationships. We also briefly discussed two types of data models: object-based data models and relation-based data models. It is a common practice in database design to develop an object-based model first and then to systematically convert the model into a relation-based model, which is more suitable for database implementation. In this chapter, we will focus on conceptual database design using object-based models. The entity-relationship model has emerged as one of the most popular techniques in the design of databases due to its inherent advantages. The entity-relationship (E-R) model is easy to learn, yet powerful enough to model complex, real-world scenarios.

We have therefore chosen the E-R model for database design discussion.

3.2 The Entity Relationship Model

An entity-relationship model describes data in terms of the following

1. Entities
2. Relationship between entities
3. Attributes of entities

We graphically display an E-R model using an entity-relationship diagram (or E-R diagram) like the sample in Figure 3.1. While this figure may seem to be confusing at first glance, its meaning should become very clear by the end of this chapter.

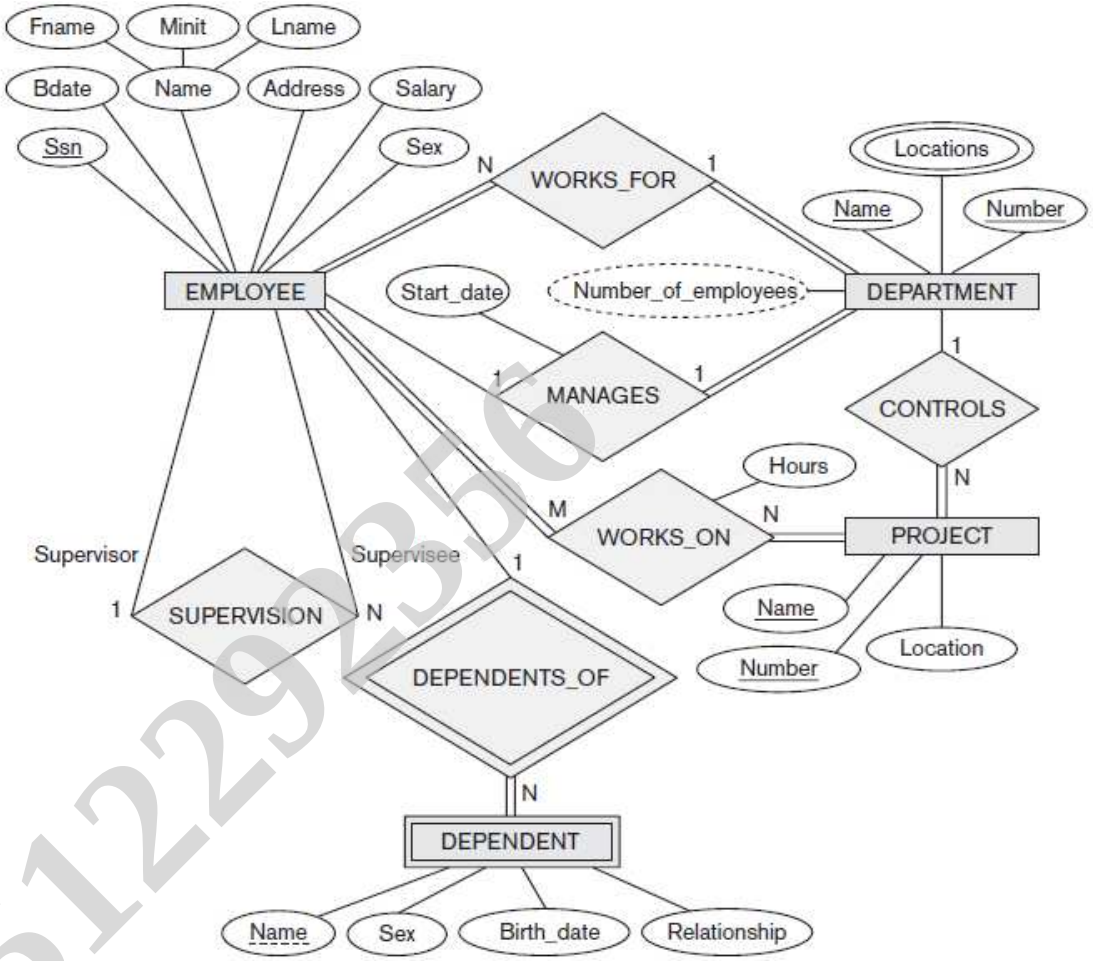


Figure 3.1: Example of an E-R diagram.

3.2.1 Entity

An entity is an object that exists, and which is distinguishable from other objects. An entity can be a person, a place, an object, an event, or a concept about which an organization wishes to maintain data. The following are some examples of entities:

Person: STUDENT, EMPLOYEE, CLIENT
Object: COUCH, AIRPLANE, MACHINE
Place: CITY, NATIONAL PARK, ROOM, WAREHOUSE
Event: WAR, MARRIAGE, LEASE
Concept: PROJECT, ACCOUNT, COURSE

It is important to understand the distinction between an entity type, an entity instance, and an entity set. An entity type defines a collection of entities that have same attributes. An entity instance is a single item in this collection. An entity set is a set of entity instances. The following example will clarify this distinction: STUDENT is an entity type; a student with ID number 555-55-5555 is an entity instance; and a collection of all students is an entity set. In the E-R diagram, we assign a name to each entity type. When assigning names to entity types, we follow certain naming conventions. An entity name should be a concise singular noun that captures the unique characteristics of the entity type. An E-R diagram depicts an entity type using a rectangle with the name of the entity inside (see Figure 3.2).

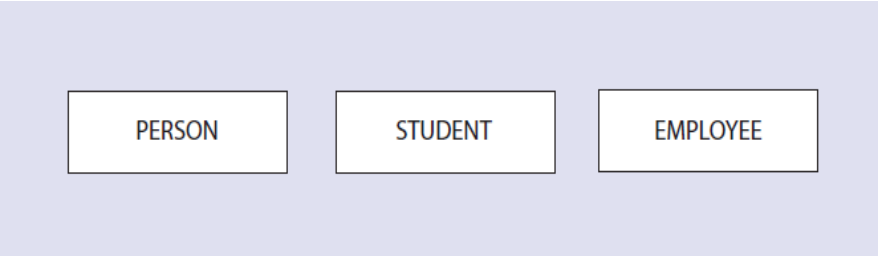


Figure 3.2: The entity representation in an E-R diagram

3.2.2 Attributes

We represent an entity with a set of attributes. An attribute is a property or characteristic of an entity type that is of interest to an organization. Some attributes of common entity types include the following:

STUDENT = {Student ID, SSN, Name, Address, Phone, Email, DOB}
ORDER = {Order ID, Date of Order, Amount of Order}
ACCOUNT = {Account Number, Account Type, Date Opened, Balance}
CITY = {City Name, State, Population}

We use the following conventions while naming attributes:

1. Each word in a name starts with an uppercase letter followed by lower case letters.
2. If an attribute name contains two or more words, the first letter of each subsequent word is also in uppercase, unless it is an article or preposition, such as “a,” “the,” “of,” or “about” (see Figure 3.3).

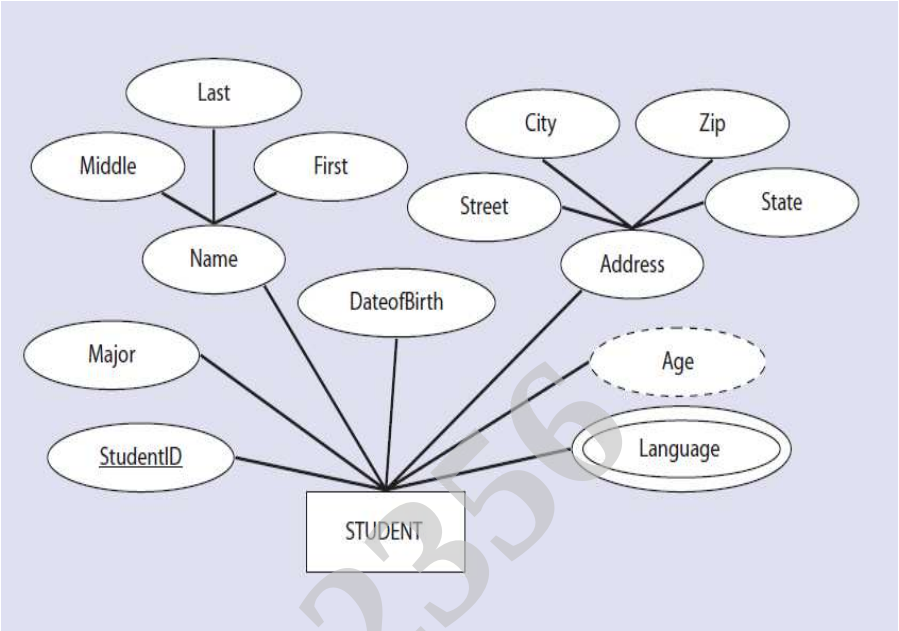


Figure 3.3: Attributes of the STUDENT entity type

E-R diagrams depict an attribute inside an ellipse and connect the ellipse with a line to the associated entity type. Figure 3.3 illustrates some of the possible attributes in an E-R diagram for the entity STUDENT.

Notice that not all the attributes in Figure 3.3 are marked in the same way. There are several types of attributes featured in this figure. These include simple, composite, single-valued, multi-valued, stored, and derived attributes. In the following subsections, we discuss the distinctions between these types of attributes.

- **Simple and Composite Attributes**

A simple or an atomic attribute, such as City or State, cannot be further divided into smaller components. A composite attribute, however, can be divided into smaller subparts in which each subpart represents an independent attribute. Name and Address are the only composite attributes in Figure 3.3. All other attributes, even those that are subcategories of Name and Address, are simple attributes. The figure also presents the notation that depicts a composite attribute.

- **Single-Valued and Multi-Valued Attributes**

Most attributes have a single value for an entity instance; such attributes are called single-valued attributes. A multi-valued attribute, on the other hand, may have more than one value for an entity instance. Figure 3.3 features one multi-valued attribute, Languages, which stores the names of the languages that a student speaks. Since a student may speak several languages, it is a multi-valued attribute. All other attributes of the STUDENT entity type are single-valued attributes. For example, a student has only one date of birth and one student identification number. In the E-R diagram, we denote a multi-valued attribute with a double-lined ellipse. Note that in a multi-

valued attribute, we always use a double-lined ellipse, regardless of the number of values.

- **Stored and Derived Attributes**

The value of a derived attribute can be determined by analyzing other attributes. For example, in Figure 3.3 Age is a derived attribute because its value can be derived from the current date and the attribute DateofBirth. An attribute whose value cannot be derived from the values of other attributes is called a stored attribute. As we will learn, a derived attribute Age is not stored in the database. Derived attributes are depicted in the E-R diagram with a dashed ellipse.

- **Key Attribute**

A key attribute (or identifier) is a single attribute or a combination of attributes that uniquely identify an individual instance of an entity type. No two instances within an entity set can have the same key attribute value. For the STUDENT entity shown in Figure 3.3, StudentID is the key attribute since each student identification number is unique. Name, by contrast, cannot be an identifier because two students can have the same name. We underline key attributes in an E-R diagram (also see Figure 3.4).

Sometimes no single attribute can uniquely identify an instance of an entity type. However, in these circumstances, we identify a set of attributes that, when combined, is unique for each entity instance. In this case the key attribute, also known as **composite key**, is not a simple attribute, but a composite attribute that uniquely identifies each entity instance.

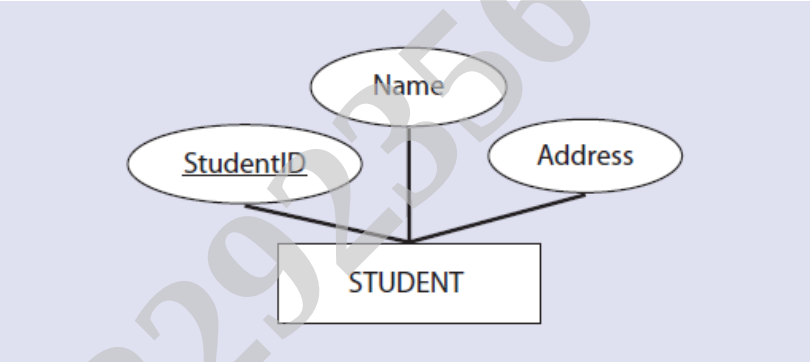


Figure 3.4: the Key attribute

A composite key must be minimal in the sense that no subset of a composite key can form the key of the entity instance. For example, if a composite key has four attributes, A1 to A4, then any subset, say A2, A4 or A2, A3 (or any of 16 combinations), should not form a key for an entity. In other words, we need all attributes, A1–A4, to identify each instance of an entity uniquely. In the E-R diagram, we underline each attribute in the composite key. For example, consider the CITY entity type (see Figure

3.5). This category includes, potentially, all the cities in the United States. Notice that none of the attributes (i.e. Name, State or Population) can serve as a key attribute since there are many cities in each state and two cities could possibly have the same name or population. However, the composite attribute {Name, State} is a valid key attribute for the CITY entity as no two cities within a state can have the same name.

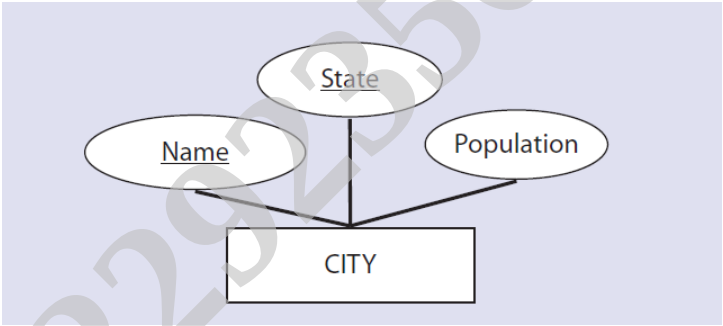


Figure 3.5: the Composite Key attribute

3.2.3 Relationships

Entities in an organization do not exist in isolation but are related to each other. Students take courses and each STUDENT entity is related to the COURSE entity. Faculty members teach courses, and each FACULTY entity is also related to the COURSE entity. Consequently, the STUDENT entity is related to the FACULTY entity through the

COURSE entity. E-R diagrams can also illustrate relationships between entities.

We define a relationship as an association among several entities. Consider, for example, an association between customers of a bank. If customer Williams has a bank account number 523, then the quality of ownership constitutes a relationship instance that associates the CUSTOMER instance Williams with the ACCOUNT instance 523. We can think of the relationship instance as a verb that links a subject and an object: customer Williams has an account; student John registers for a course; Professor Smith teaches a course. A relationship set is a grouping of all matching relationship instances, and the term relationship type refers to the relationship between entity types. For example, Figure 3.6 illustrates a relationship set between the CUSTOMER and the ACCOUNT instances.

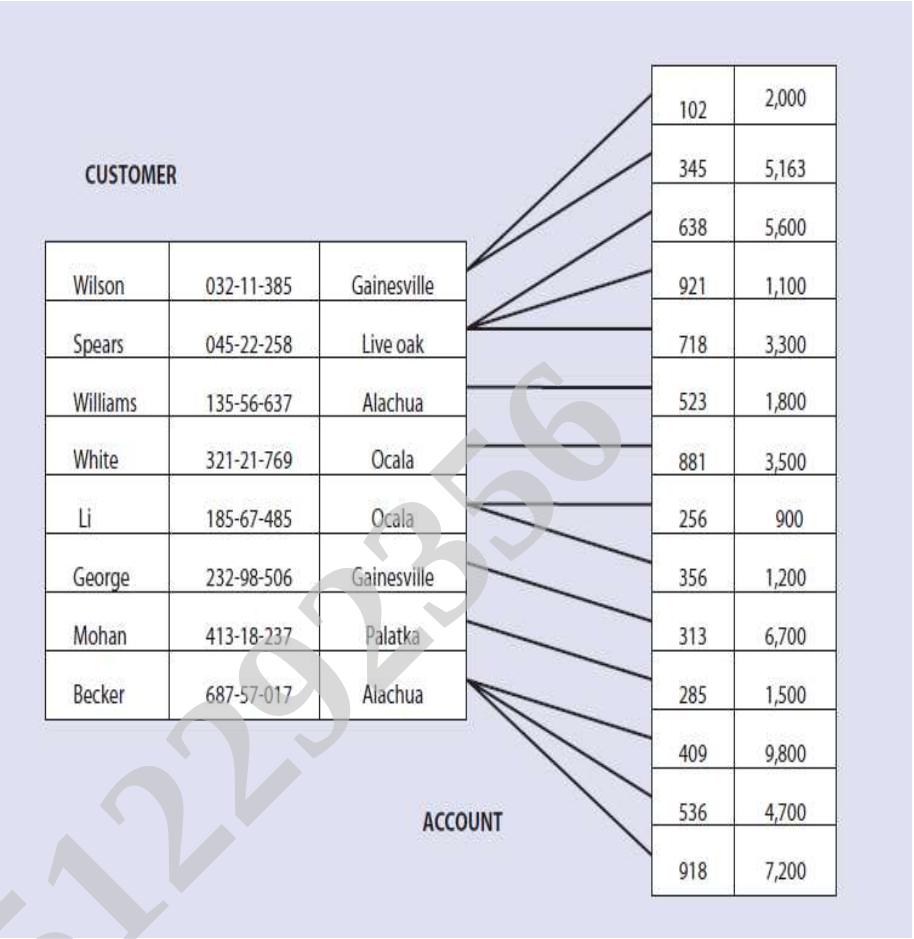


Figure 3.6: The relationship set between the CUSTOMER and ACCOUNT entities.

In an E-R diagram, we represent relationship types with diamond-shaped boxes connected by straight lines to the rectangles that represent participating entity types. A relationship type is a given name that is displayed in this diamond-shaped box and typically takes the form of a

present tense verb or verb phrase that describes the relationship. An E-R diagram may depict a relationship as the previous example of the relationship between the entities CUSTOMER and ACCOUNT does:

3.3 Cardinality of Relationship

The term cardinal number refers to the number used in counting. An ordinal number, by contrast, emphasizes the order of a number (1st, 7th, etc.). When we say cardinality of a relationship, we mean the ability to count the number of entities involved in that relationship. For example, if the entity types **A** and **B** are connected by a relationship, then the **maximum cardinality** represents the maximum number of instances of entity B that can be associated with any instance of entity **A**.

However, we don't need to assign a number value for every level of connection in a relationship. In fact, the term maximum cardinality refers to only two possible values: one or many. While this may seem to be too simple, the division between one and many allows us to categorize all of the permutations possible in any relationship. The maximum cardinality value of a relationship, then, allows us to define the four types of relationships possible between entity types A and B. Figure 3.7 illustrates these types of relationships.

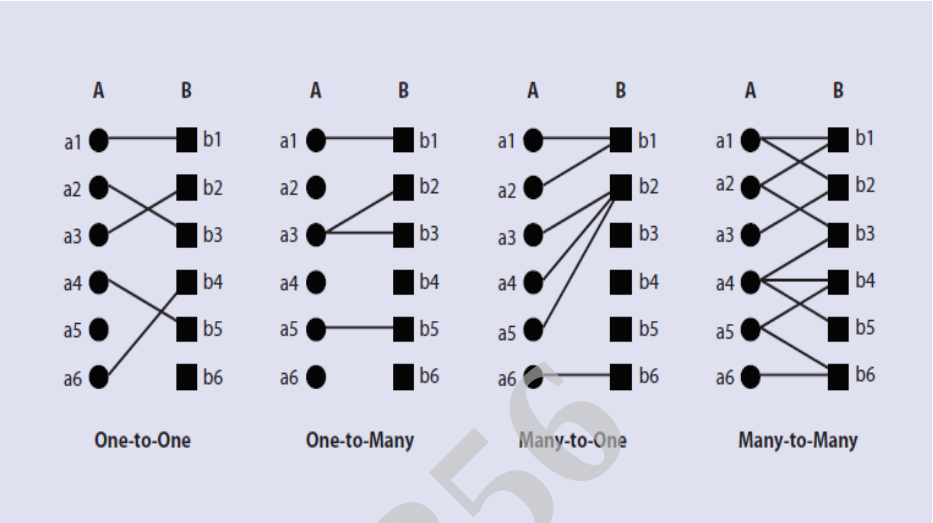


Figure 3.7: The four types of relationships between entity types A and B.

One-to-One Relationship In a one-to-one relationship, at most one instance of entity **B** can be associated with a given instance of entity **A** and vice versa.

One-to-Many Relationship In a one-to-many relationship, many instances of entity **B** can be associated with a given instance of entity **A**. However, only one instance of entity **A** can be associated with a given instance of entity **B**. For example, while a customer of a company can makes many orders, an order can only be related to a single customer.

Many-to-Many Relationship In a many-to-many relationship, many instances of entity **A** can be associated

with a given instance of entity **B**, and, likewise, many instances of entity **B** can be associated with a given instance of entity **A**. For example, a machine may have different parts, while each individual part may be used in different machines.

Representing Relationship Types Figure 3.8 displays how we represent different relationship types in an E-R diagram. An entity on the one side of the relationship is represented by a vertical line, "I," which intersects the line connecting the entity and the relationship. Entities on the many side of a relationship are designated by a crowfoot as depicted in Figure 3.8.

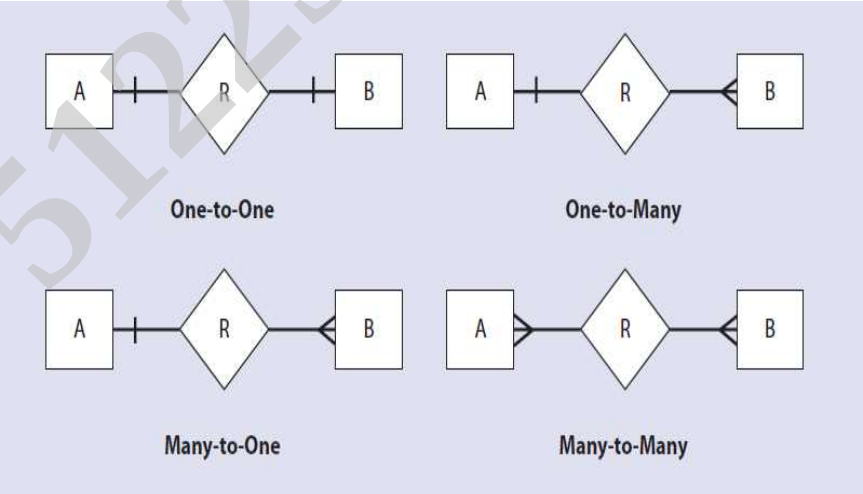


Figure 3.8: The relationship types based on maximum cardinality

We will now discuss the **minimum cardinality** of a relationship. The minimum cardinality between two entity types A and B is defined as the minimum number of instances of entity B that must be associated with each instance of entity A. In an E-R diagram, we allow the minimum cardinality to take two values: zero or one. If the minimum cardinality is zero, we say that entity type B is an optional participant in the relationship; otherwise, it is a mandatory participant. An optional relationship is represented by an "O" and mandatory relationship is represented by "|" in an E-R diagram.

Figure 3.9 shows the four possibilities of the minimum cardinality of a relationship between two entity types A and B. Figure 3.9(a) depicts a situation in which no minimum cardinality constraints exist between the instances of entities A and B, meaning both entities A and B are optional participants in the relationship. Figure 3.9(b) illustrates a situation in which each instance of entity B must be associated with at least one instance of entity A, but no association is required for an instance of entity A. Figure 3.9(c) illustrates a situation in which each instance of entity A must be associated with at least one instance of entity B, but no association is required for an instance of entity B. Finally, Figure 3.9(d) illustrates a situation in which each instance of entity A and B must be

associated with at least one instance of entity B and A, respectively.

An E-R diagram displays both the maximum and the minimum cardinalities of the relationships between two entities. Since there are four basic possibilities of maximum cardinalities and four possibilities of minimum cardinalities between two entities, there are 16 types of relationships possible between two entities in terms of cardinality. We will see several examples of these relationships while studying unary, binary, and ternary relationships.

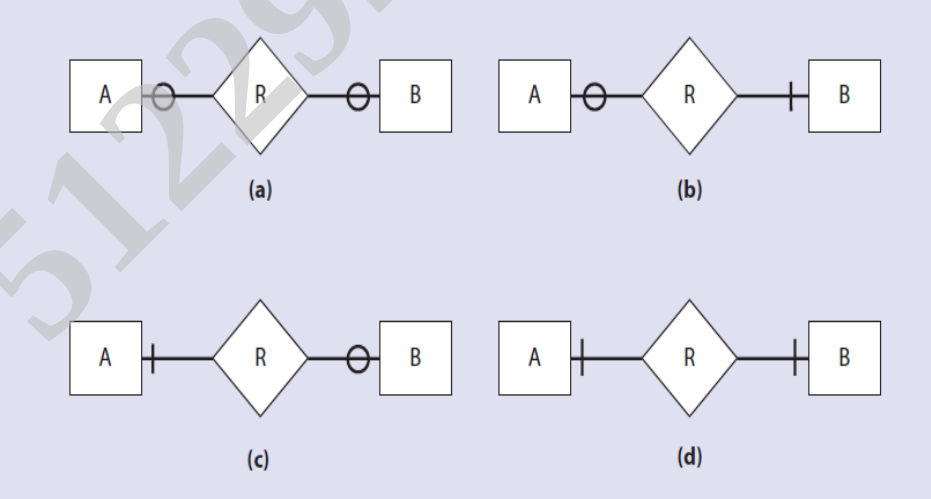


Figure 3.9: The relationship types based on minimum cardinality

3.4 Degree of Relationship

The number of entities sets that participate in a relationship is called the **degree of relationship**. For example, the degree of the relationship featured in Figure 3.10 is two because CUSTOMER and ACCOUNT are two separate entity types that participate in the relationship. The three most common degrees of a relationship in a database are unary (degree 1), binary (degree 2), and ternary (degree 3). We will briefly define these degrees and then explore each kind of relationship in detail in subsequent sections.

3.4.1 Unary Relationship

A unary relationship ($R \in E_1 \times E_1$) is an association between two entities of the same entity type. Figure 3.10(a) displays the E-R diagram of a unary relationship IsMarriedTo. Whenever two people in the entity type PERSON get married, the relationship instance IsMarriedTo is created. The Date of marriage is an attribute of this relationship. Since a person can only be married to one other person, marriage is a one-to-one relationship. Furthermore, since a person can be unmarried, the minimum cardinality of the IsMarriedTo relationship is zero.

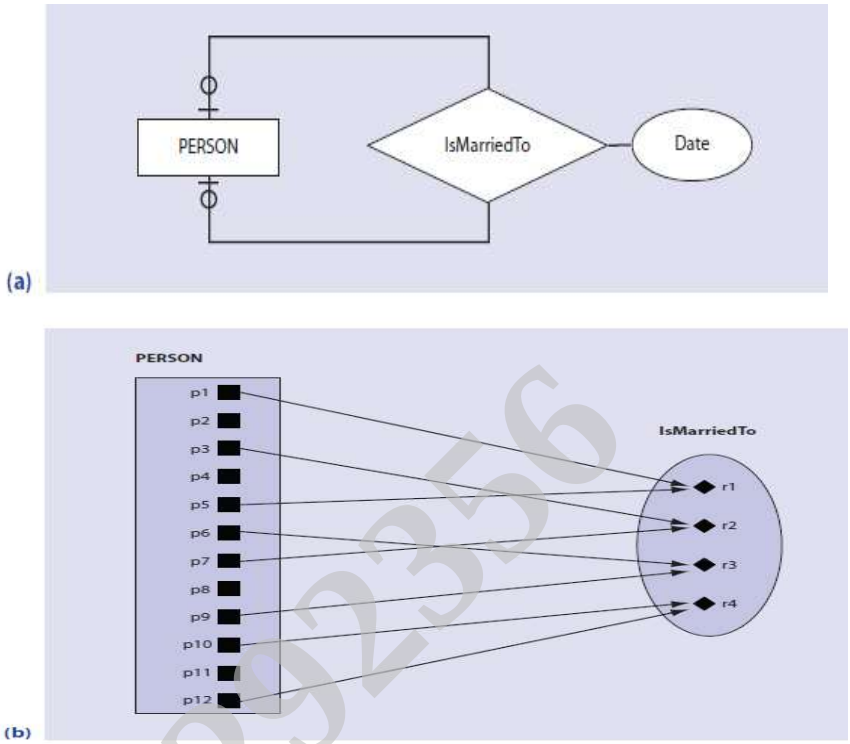


Figure 3.10: The unary one-to-one relationship.

Figure 3.10(b) depicts several relationship instances of this relationship type. Each relationship instance (r1, r2, r3, and r4) connects two instances in PERSON. The lines allow us to read relationships between entity instances. For example, r_1 suggests that person p_1 is married to person p5, and so forth.

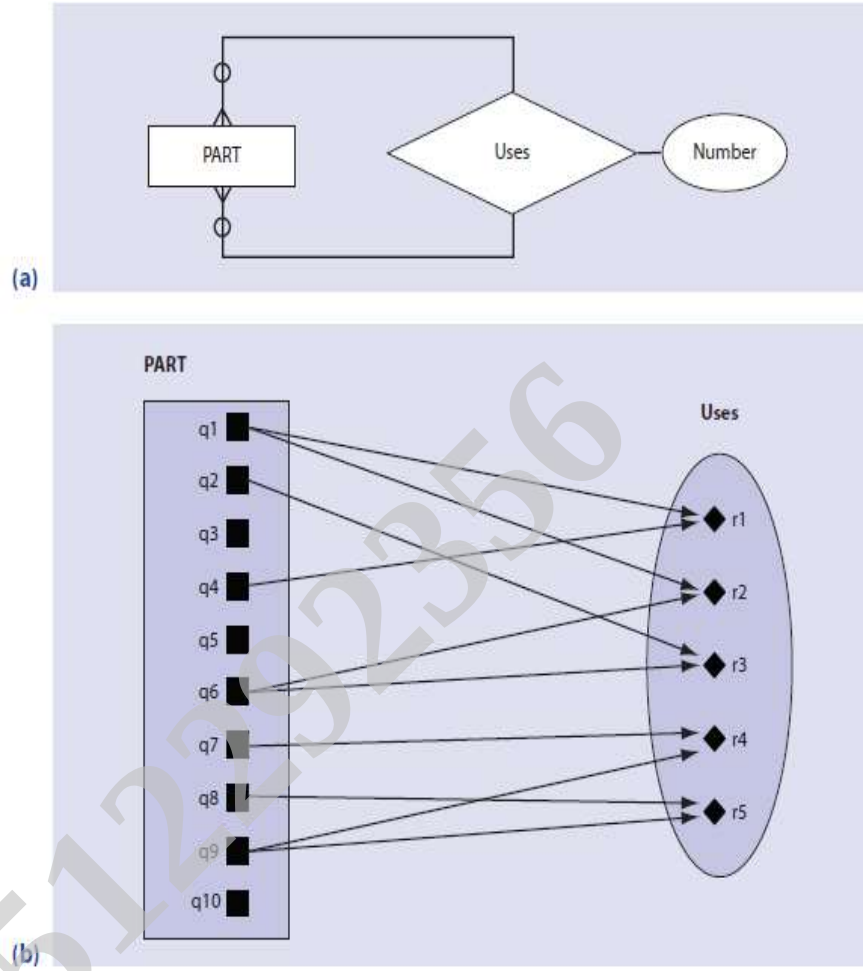


Figure 3.11: The unary many-to-many relationship.

Figure 3.11(a) provides an example of a unary relationship in which certain parts are used to make other parts. The attribute Number stores the number of parts used. The relationship uses quantifies the number of parts used to make any other part. The Uses relationship is a many-to-

many relationship since a part can use many parts and can be used in the making of many parts. The minimum cardinality of this relationship is zero because a part may not use any other part and may not be used by any other part. Figure 3.11(b) displays five instances of this relationship. Observe that several lines can emanate from a part instance and several lines terminate at a part instance.

3.4.2 Binary Relationship

A binary relationship, is an association between two entity types, is the most common form of a relationship expressed by an E-R diagram. Recall that a binary relationship is $R \in E1 \times E2$ in which E1 and E2 are two different entity types. The examples of binary relationships with the relationship instances are presented in Figure 3.12. Notice that each relationship instance obeys a basic characteristic of the binary relationships; in other words, each relationship instance is connected to exactly two entity instances of different entity types. By applying what we have learned earlier in this chapter, we should be able to determine the cardinalities of these relationships quite easily. The relationship in Figure 3.12(a) is a one-to-one relationship since we assume that an employee can manage at most one department and each department is managed by at

most one employee. The minimum cardinality can be determined since each department must be managed by an employee, but not all employees manage departments. For example, while some of the employees in Figure 3.12(b), such as e2, e4, and e6, do not manage a department, every department is managed by an employee.

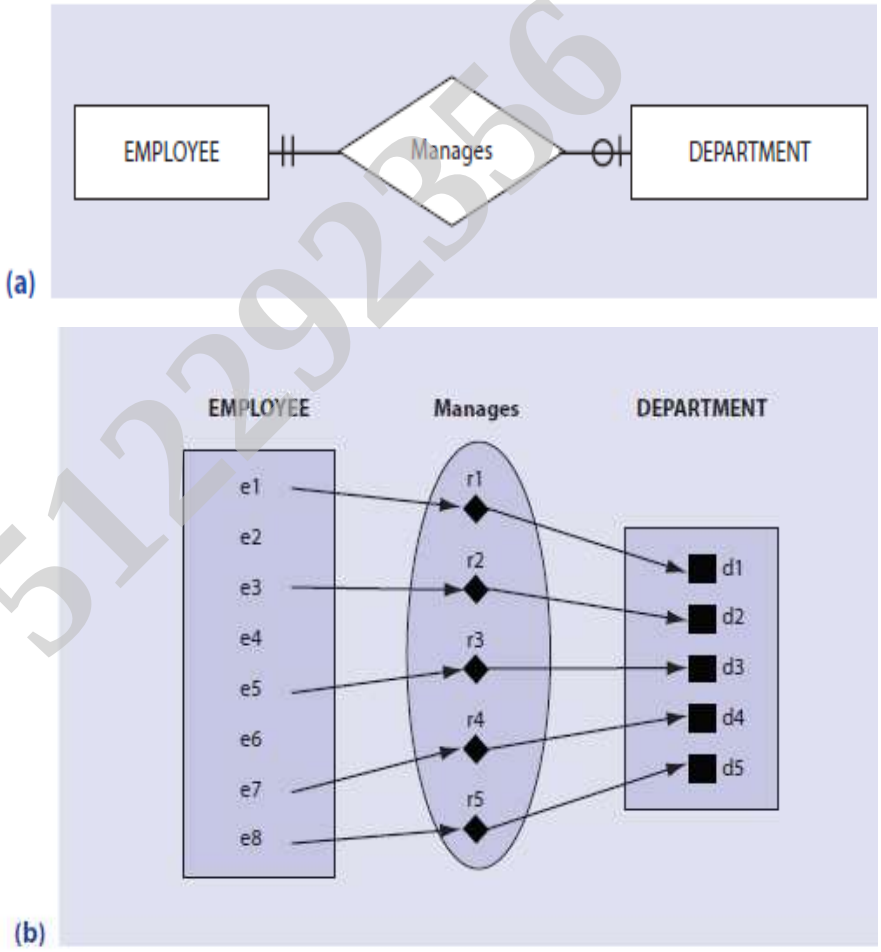


Figure 3.12: The Binary one-to-one relationship.

The binary one-to-many relationship represented in Figure 3.13(c) features a slightly different arrangement. While a customer can place several orders or may choose not to order at all, each order must be placed by exactly one customer. Figure 3.13(d) shows us that, while each order is made by a single customer, not all customers place orders.

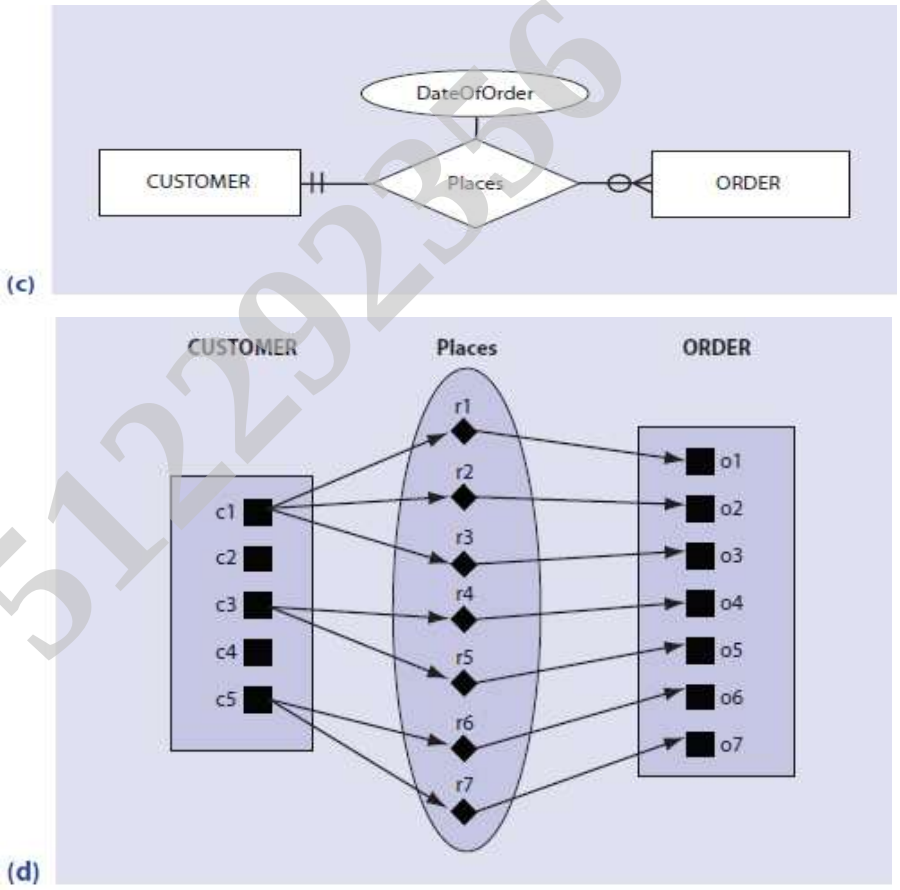


Figure 3.13: The Binary one-to-many relationship

3.4.3 Ternary Relationship

Recall that a ternary relationship R is a relationship among instances of three different entity types, E1, E2, and E3 (R ∈ E1 × E2 × E3). Each instance of the ternary relationship R requires the participation of an instance from each of the entity types E1, E2, and E3. See Figure 3.14 for examples of ternary relationships. In Figure 3.14(c), students use equipment to work on projects; each instance of Uses involves an instance of STUDENT, PROJECT, and EQUIPMENT. If a student uses two pieces of equipment to work on a project, there are two instances of the relationship Uses. A campus lab may use the attribute in this ternary relationship, the Date of use, to log the equipment usage. Ternary relationships differ significantly from the other kinds of relationships that we have examined so far. It is important to remember that a ternary relationship is not equivalent to two binary relationships.

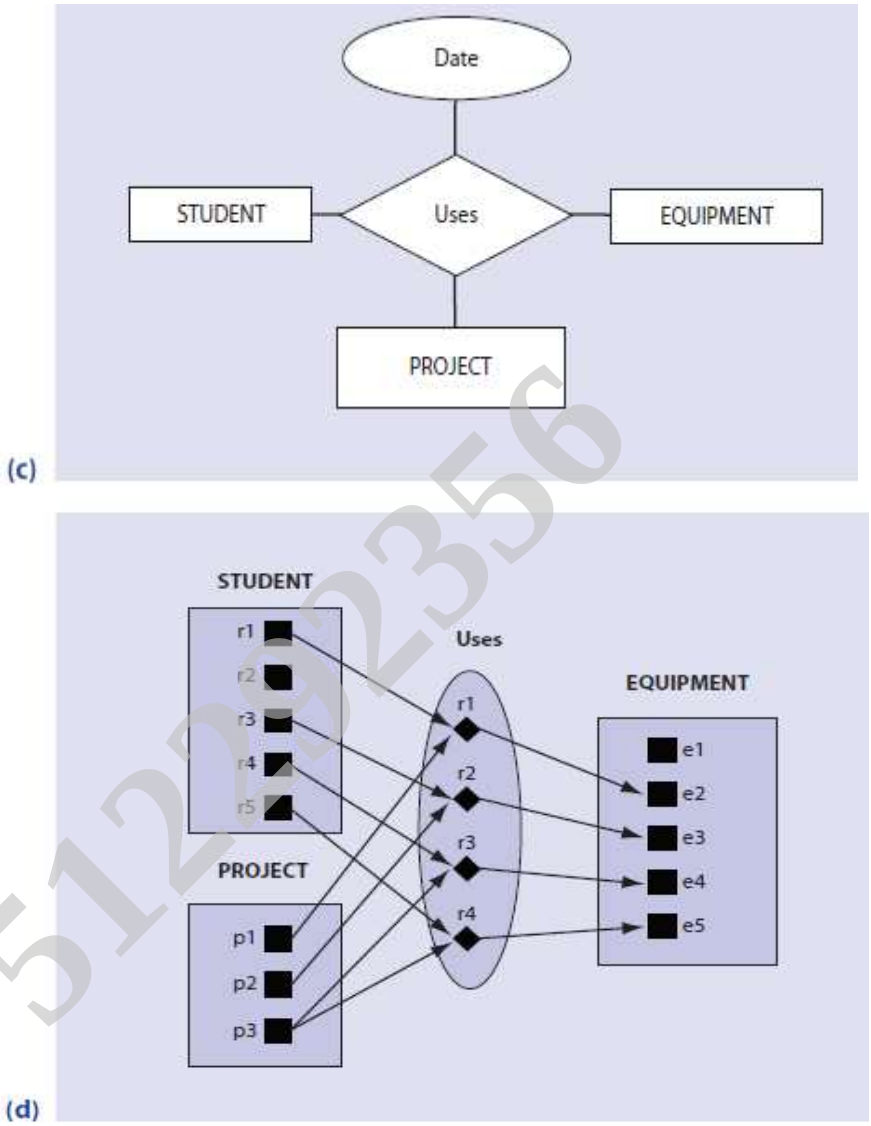


Figure 3.14: The Ternary relationship

3.5 Attributes of Relationship

We have already discussed examples of E-R diagrams that reveal attributes stemming off relationships. For example.

In Figure 3.15. In E-R diagram, the attribute DateofOrder collects data for the relationship CUSTOMER places ORDER. Attributes on relationships are like attributes on entity types we have seen so far. An attribute on a relationship stores information related to the relationship. In Figure 3.16, the attribute Quantity stores the number of components that make up an entity type ITEM. Note that the attribute Quantity stems from the relationship and not from the entity.

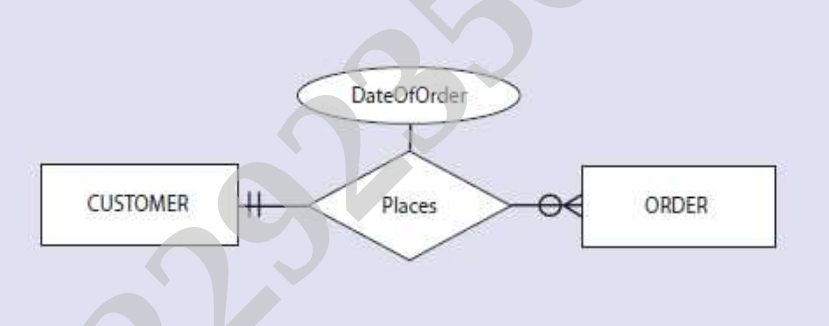


Figure 3.15: An example of attributes of binary relationships.

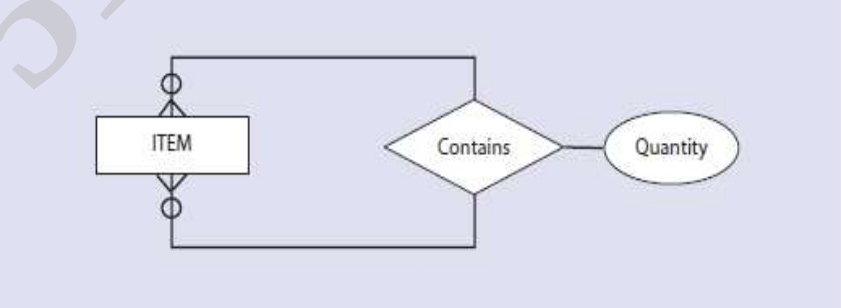


Figure 3.16: An example of attributes of unary relationships.

3.6 Associative Entities

An associative entity is an entity type that connects the instances of one or more entity types and contains attributes particular to this association. Basically, an associative entity is a relationship that has been turned into an entity because it meets one of the following conditions:

1. It is a many-to-many binary relationship
2. It is a ternary relationship or a relationship of an even higher degree

The associative entity in an E-R diagram is represented by an entity box enclosing the diamond relationship symbol (see Figure 3.17). This symbol demonstrates that the entity is generated from a relationship.

Consider, for example, the E-R diagram of the binary relationship illustrated in Figure 3.17(a). When we convert this relationship into an associated entity, we get the E-R diagram in Figure 3.17(b). In the example, the relationship Participates is converted into an associated entity,

ENROLLMENT. If the relationship has attributes, they become the attributes of the corresponding associative

entity. DateJoined is an example of an attribute of a relationship turned into an attribute of an associative entity in Figure 3.17(b).

Furthermore, recall that every entity in an E-R diagram must have an identifier. The identifiers of two original entities together serve as a composite identifier of the associative entity. In our example, the identifier of the entity ENROLLMENT is a composite attribute comprised of the identifiers of the STUDENT and ACTIVITY entities. Since a student can enroll in every activity but can only enroll once in any given activity, the composite attribute {StudentID, ActivityID} is a valid identifier for the associated entity ENROLLMENT.

Now that ENROLLMENT is a new entity in Figure 3.17 (b), note that there is no relationship diamond on the line between the entity STUDENT and the associative entity ENROLLMENT because the associative entity represents a relationship. Furthermore, the E-R diagram in Figure 3.17 (b) depicts the cardinalities. Each instance of the STUDENT entity is related to several instances of the ENROLLMENT entity; in fact, each instance of the STUDENT entity is related to as many instances of the ENROLLMENT entity as the number of activities in which the student participates. Therefore, there is a one-to-many relationship between the STUDENT entity type and the

ENROLLMENT entity type. Similarly, there is a one-to-many relationship between the entity types of ACTIVITY and ENROLLMENT because several students may enroll in one activity.

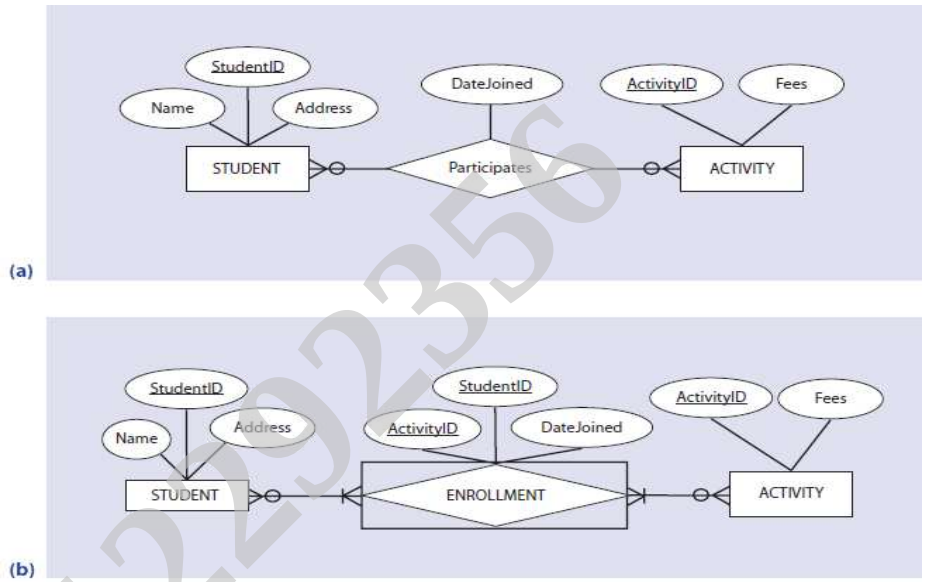


Figure 3.17: (a) A many-to-many binary relationship; (b) A many-to-many binary relationship converted to an associated entity.

Notice that when we convert a relationship into an associative entity, we change its name from a verb to a noun. Therefore, the relationship expressed by the verb participates in Figure 3.17(a) becomes the associative entity represented by the noun enrollment in Figure 3.17(b). This grammatical distinction parallels the main

reason for turning a relationship into an associative entity. A verb expresses an action, a process. However, once we want to count, record, or analyze an instance of that process, we end up with a person, place, or thing. In short, by transforming a relationship into an associative entity, we can analyze and record a process in the same way that we analyze and record a noun.

3.7 Weak Entity Types

Entity types can be classified into two categories: strong entity types and weak entity types. A strong entity type exists independent of other entity types, while a weak entity type depends on another entity type. Consider an employee database that includes an entity EMPLOYEE. Suppose that we also record data about each employee's dependents, such as a wife or children, in this database. To do so, we must create the entity type DEPENDENT. The entity type DEPENDENT does not exist on its own and owes its existence to the EMPLOYEE entity type. When employee's record are removed from the EMPLOYEE entity set, the records of their dependents are also removed from the DEPENDENT entity set. In the E-R diagram, a weak entity is indicated by a double-lined rectangle. The corresponding relationship diamond is also double-lined. The entity type on which a weak entity type depends is called the identifying owner (or simply owner),

and the relationship between a weak entity type and its owner is called an identifying relationship. In Figure 3.18, EMPLOYEE is the owner and *has* is the identifying relationship. For a weak entity, we define a partial key attribute that, when combined with the key attribute of its owner, provides the full identifier for the weak entity. In Figure 3.18, for example, DependentName is the partial identifying attribute. When we combine it with the EmployeeID, it uniquely identifies the dependent. Of course, in this example, we make an implicit assumption that people do not give the same name to more than one of their children.

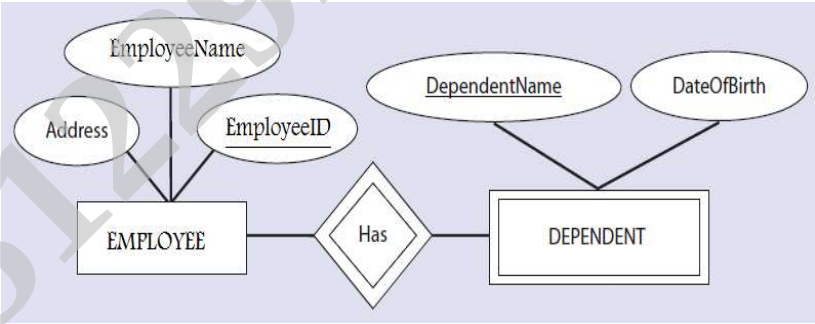


Figure 3.18: The weak entity in an E-R diagram

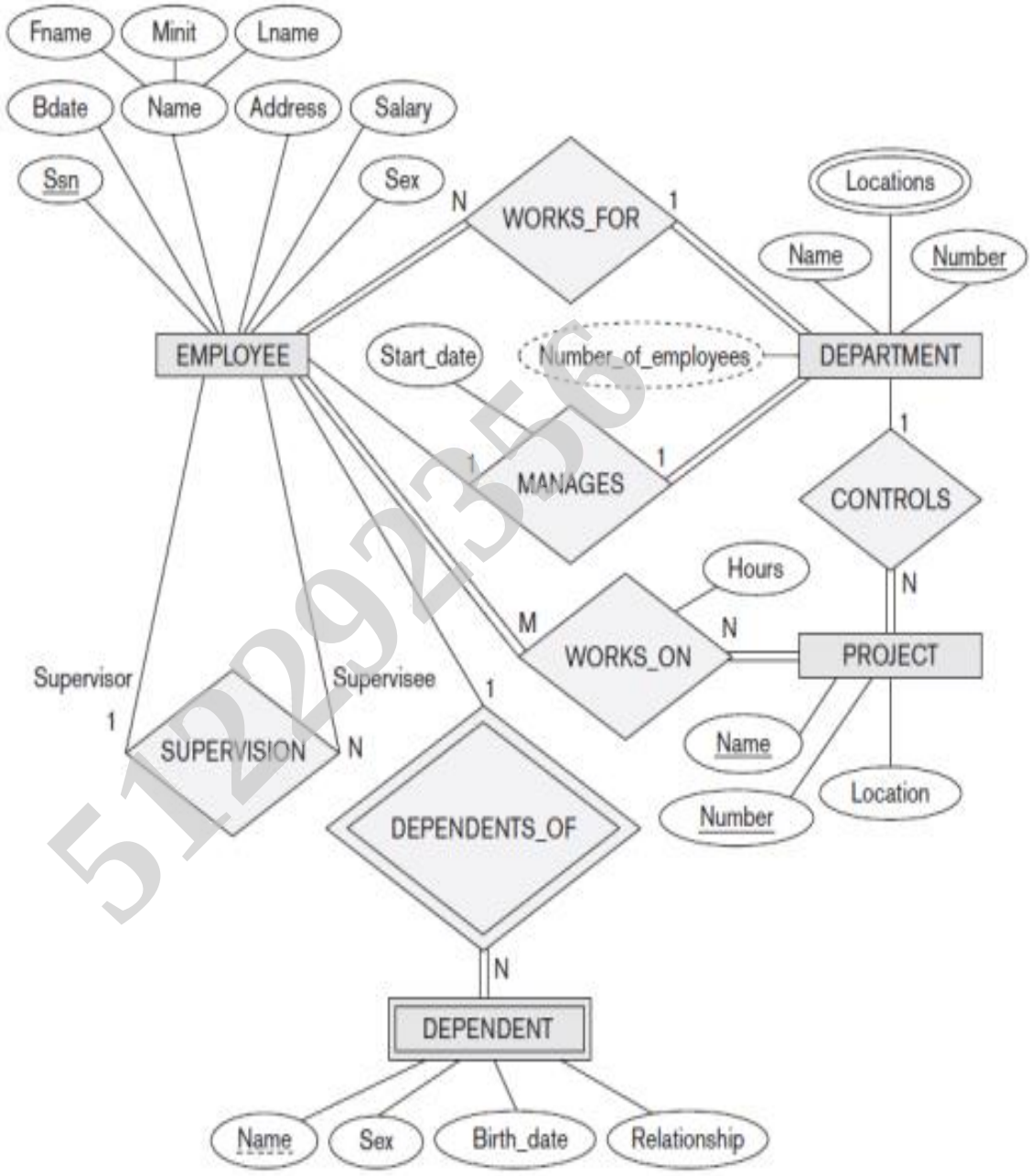
Chapter 4 Relational Database Design

4.1 Introduction

ER Modeling used to understand the informational needs of a system. Once the model is satisfactory, we then implement our design in a relational database. In order to implement a database, we must decide on the relations, their attributes and primary keys. This chapter contains some simple rules that we use to map an ERD to a relational database. In general, relations are used to hold entity sets and to hold relationship sets.

ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, but an approximate schema can be generated.

There are several processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual. We may focus here on the mapping diagram contents to relational basics. We will discuss all cases for transforming the ER Model into Relational Schema via the following ER Model for company


Fname
Minit
Lname
Bdate
Name
Address
Salary
Ssn
Sex
N
WORKS_FOR
1
Locations
Name
Number
EMPLOYEE
Start_date
Number_of_employees
DEPARTMENT
1
1
MANAGES
1
CONTROLS
Hours
N
M
N
WORKS_ON
PROJECT
Supervisor
1
Supervisee
1
SUPERVISION
N
Name
Location
Number
DEPENDENTS_OF
N
DEPENDENT
Name
Sex
Birth_date
Relationship

STEP 1: MAP REGULAR ENTITY TYPES

For each regular entity type, create a relation schema R that includes all the single-valued attributes of E

• composite attributes are divided into simple attributes.

• Example renames some attributes (e.g., Dname), but not needed

• Pick one of the keys as “primary key” and declare the rest to be unique

• Called entity relations

• Each tuple represents an entity instance

(a) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary

DEPARTMENT

Dname	Dnumber

PROJECT

Pname	Pnumber	Plocation

STEP 2: MAP WEAK ENTITY TYPES

For each weak entity type, create a relation schema R and include all single-valued attributes of the weak entity type and of the identifying relationship as attributes of R

• Include primary key attribute of identifying entity as foreign key attribute of R

• Primary key of R is primary key of identifying entity together with partial key from R

• Remove the identifying relationship when subsequently translating (other) relationship types to relation schemas

(b) **DEPENDENT**

Essn	Dependent_name	Sex	Bdate	Relationship

STEP 3: MAP BINARY 1:1 RELATIONSHIP TYPES

For each binary 1:1 relationship type R, identify relation schemas that correspond to entity types participating in R

• Apply one of three possible approaches:

- Foreign key approach
  - Add primary key of one participating relation as foreign key attribute of the other, which will also represent R

- If only one side is total, choose it to represent R (why?)
- Declare foreign key attribute as unique

- Merged relationship approach
  - Combine the two relation schemas into one, which will also represent R
  - Make one of the primary keys “unique” instead

- Cross-reference or relationship relation approach
  - Create new relation schema for R with two foreign key attributes being copies of both primary keys
  - Declare one of the attributes as primary key and the other one as unique

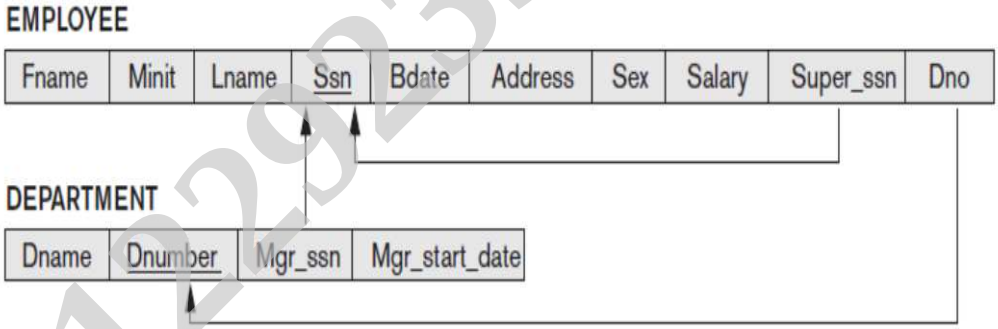
- Add single-valued attributes of relationship type as attributes of R

STEP 4: MAP BINARY 1:N

- Foreign key approach
  - Identify relation schema S that represents participating entity type at N-side of 1:N relationship type

• Include primary key of other entity type (1-side) as foreign key in S

- Relationship relation approach
  • Create new relation schema for S with two foreign key attributes being copies of both primary keys
  • Declare the foreign key attribute for the relation schema corresponding to the participating entity type on the N-side as primary key
- Include single-valued attributes of relationship type as attributes of S



STEP 5: MAP BINARY M:N

For each binary M:N relationship type or ternary or higher order relationship type, create a new relation S

• Include primary key of participating entity types as foreign key attributes in S

• Make all these attributes primary key of S

• Include any simple attributes of relationship type in S

(c) WORKS_ON

Essn	Pno	Hours

STEP 6: MAP MULTIVALUED ATTRIBUTES

For each multivalued attribute

• Create new relation R with attribute to hold multivalued attribute values

• If multivalued attribute is composite, include its simple components

• Add attribute(s) for primary key of relation schema for entity or relationship type to be foreign key for R

• Primary key of R is the combination of all its attributes

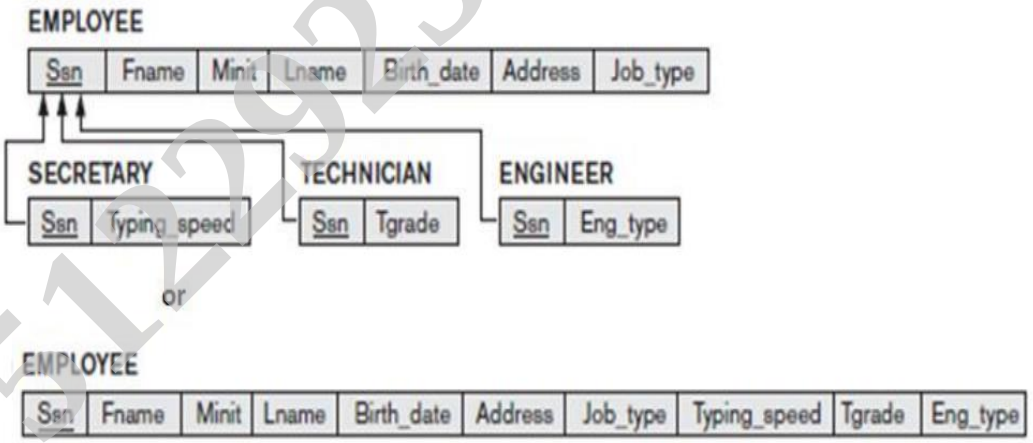
(d) DEPT_LOCATIONS

Dnumber	Dlocation

OPTIONS FOR MAPPING SPECIALIZATION OR GENERALIZATION

For any specialization (total or partial, disjoint or overlapping)

- Separate relation per superclass and subclasses
- Single relation with at least one attribute per subclass
- Introduce a Boolean attribute if none specific for subclass



For total specializations (and generalizations) only

- Separate relation per subclass relations only
- Overlapping subclasses will result in multiple tuples per entity

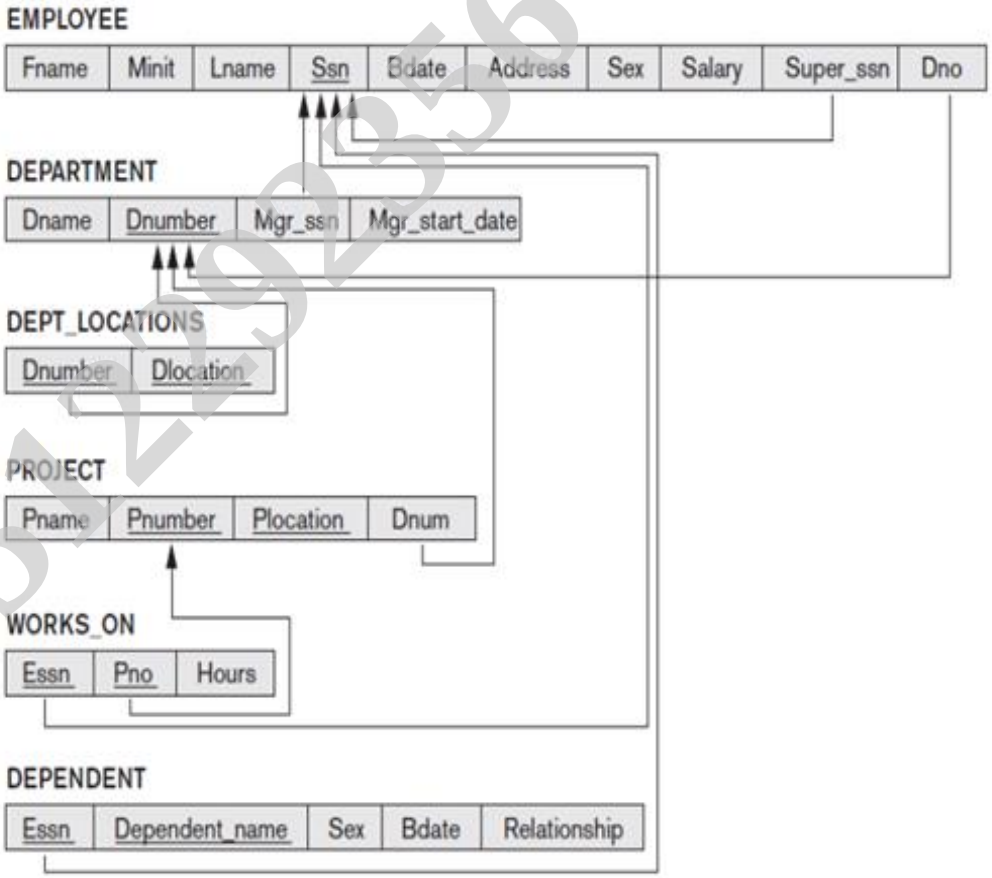
♣ For disjoint specializations only

• Single relation with one type of attribute

• Type or discriminating attribute indicates subclass of tuple

• Might require many NULL values if several specific attributes exist in subclasses

The Full Solution



Chapter 5
SQL Basics

5.1 introduction

SQL stands for Structured Query Language. SQL is a standard programming language specifically designed for storing, retrieving, managing, or manipulating the data inside a relational database management system (RDBMS). SQL became an ISO standard in 1987.

SQL is the most widely implemented database language and supported by the popular relational database systems, like MySQL, SQL Server, and Oracle. However, some features of the SQL standard are implemented differently in different database systems.

SQL was originally developed at IBM in the early 1970s. Initially it was called SEQUEL (Structured English Query Language) which was later changed to SQL (pronounced as S-Q-L).

What You Can Do with SQL There are lot more things you can do with SQL:

1. You can create a database.

2. You can create tables in a database.

3. You can query or request information from a database.

4. You can insert records in a database.

5. You can update or modify records in a database.

6. You can delete records from the database.

7. You can set permissions or access control within the database for data security.

8. You can create views to avoid typing frequently used complex queries. SQL Syntax

SQL follows some unique set of rules and guidelines called syntax. Here, we are providing all the basic SQL syntax.

1. SQL is not case sensitive. Generally, SQL keywords are written in uppercase.

2. SQL statements are dependent on text lines. We can place a single SQL statement on one or multiple text lines.

3. You can perform most of the action in a database with SQL statements.

4. SQL depends on relational algebra and tuple relational calculus.

SQL statement

SQL statements are started with any of the SQL commands/keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP etc. and the statement ends with a semicolon (;).

Example of SQL statement:

SELECT "column name" FROM "table name".

SQL Commands

Structured Query Language (SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like Create, Drop, and Insert etc. to carry out the required tasks.

These SQL commands are mainly categorized into four categories as:

- DDL — Data Definition Language.
- DQ1 — Data Query Language.
- DML — Data Manipulation Language.
- DCL — Data Control Language.
- TCL — Transaction Control Language

DDL (Data Definition Language). DDL or Data Definition Language consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

- CREATE — is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- DROP — is used to delete objects from the database.
- ALTER-is used to alter the structure of the database.
- TRUNCATE—is used to remove all records from a table, including all spaces allocated for the records are removed.
- COMMENT —is used to add comments to the data dictionary.
- RENAME —is used to rename an object existing in the database.

DQL (Data Query Language):

DQL statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.

Example of DQL:

• SELECT — is used to retrieve data from the database.

DML (Data Manipulation Language): The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Examples of DML:

1. INSERT — is used to insert data into a table.

2. UPDATE — is used to update existing data within a table.

3. DELETE — is used to delete records from a database table.

DCL (Data Control Language): DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions, and other controls of the database system.

Examples of DCL commands:

■ GRANT-gives user's access privileges to database.

■ REVOKE-withdraw user's access privileges given by using the GRANT command.

TCL (transaction Control Language): TCL commands deals with the transaction within the database.

Examples of TCL commands:

- COMMIT— commits a Transaction.

- ROLLBACK— rollbacks a transaction in case of any error occurs.
- SAVEPOINT—sets a save point within a transaction.
- SET TRANSACTION—specify characteristics for the transaction.

SQL Data types

Like in other programming languages, SQL also has certain data types available. A brief idea of all the data types is discussed below.

Data Type	Description
binary	Maximum length of 8000 bytes (Fixed-Length binary data)
varbinary	Maximum length of 8000 bytes(Variable Length binary data)
varbinary(max)	Maximum length of 231 bytes (SQL Server 2005 only). (Variable Length binary data)
image	Maximum length of 2,147,483,647 bytes(Variable Length binary data)

- Exact Numeric Data type:

There are nine subtypes which are given below in the table. The table contains the range of data in a particular type.

Data Type	From	To
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	$-10^{38} + 1$	$10^{38} - 1$
numeric	$-10^{38} + 1$	$10^{38} - 1$
money	-922,337,203,685,477.5808	922,337,203,685,477.5808
smallmoney	-214,748.3648	214,748.3648

3. Approximate Numeric Data type: The subtypes of this data type are given in the table with the range.

Data Type	From	To
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

4. Character String Data type: The subtypes are given in below table

Data Type	Description
char	Maximum length of 8000 characters. (Fixed-Length non-Unicode Characters)
varchar	Maximum length of 8000 characters. (Variable-Length non-Unicode Characters)
varchar(max)	Maximum length of 231 characters (SQL Server 2005 only). (Variable Length non-Unicode data)
text	Maximum length of 2,147,483,647 characters (Variable Length non-Unicode data)

5. Unicode Character String Data type: The details are given in below table

Data Type	Description
nchar	Maximum length of 4000 characters. (Fixed-Length Unicode Characters)
Nvarchar	Maximum length of 4000 characters. (Variable-Length Unicode Characters)
nvarchar(max)	Maximum length of 231 characters (SQL Server 2005 only). (Variable Length Unicode data)

6. Date and Time Data type: The details are given in below table.

Data Type	From	To
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

In this chapter we discuss DDL Commands

SQL CREATE Database

The SQL CREATE DATABASE statement is used by a developer to create a database.

Let's see the syntax of SQL CREATE DATABASE:

```
CREATE DATABASE database_name,
```

SQL DROP Database

SQL DROP statement is used to delete or remove indexes from a table in the database.

If you want to delete or drop an existing database in a SQL schema, you can use SQL DROP DATABASE

Let's see the syntax of SQL DROP DATABASE:

```
DROP DATABASE database name.
```

SQL RENAME Database

SQL RENAME DATABASE is used when you need to change the name of your database. Sometimes it is used because you think that the original name is not more relevant to the database, or you want to give a temporary name to that database.

To rename the MySQL database, you need to follow the following syntax:

```
RENAME DATABASE olddb_name TO newdb_name,
```

SQL CREATE TABLE

SQL CREATE TABLE statement is used to create table in a database. If you want to create a table, you should name the table and define its column and each column's data type.

Let's see the simple syntax to create the table.

```
create table "tablename"
("columnl" "data type",
"column2" "data type",
"column3" "data type",
.....
"columnN" "data type");
```

The data type of the columns may vary from one database to another. For example, NUMBER is supported in Oracle database for integer value whereas INT is supported in MySQL.

Let us take an example to create a STUDENTS table with ID as primary key and NOT NULL are the constraint showing that these fields cannot be NULL while creating records in the table.

```
Create Table Students (
Id          Int          Not Null,
Name        Varchar (20)  Not Null,
Age         Int           Not Null,
Address     Char (25),
Primary Key (Id)
);
```

SQL Constraints

A constraint is simply a restriction placed on one or more columns of a table to limit the type of values that can be stored in that column. Constraints provide a standard mechanism to maintain the accuracy and integrity of the data inside a database table.

There are several different types of constraints in SQL, including:

- **NOT NULL**
- **PRIMARY KEY**
- **UNIQUE**
- **DEFAULT**
- **FOREIGN KEY**
- **CHECK**

NOT NULL Constraint

The NOT NULL constraint specifies that the column does not accept NULL values.

This means if NOT NULL constraint is applied on a column, then you cannot insert a new row in the table without adding a non-NULL value for that column.

The following SQL statement creates a table named persons with four columns, out of which three columns, id, name, and phone do not accept NULL values.

```
CREATE TABLE persons (
id INT NOT NULL,
name VARCHAR(30) NOT NULL,
birth date DATE,
phone VARCHAR( I 5) NOT NULL
);
```


PRIMARY KEY Constraint

The PRIMARY KEY constraint identifies the column or set of columns that have values that uniquely identify a row in a table. No two rows in a table can have the same primary key value. Also, you cannot enter NULL value in a primary key column.

The following SQL statement creates a table named persons and specifies the id column as the primary key. That means this field does not allow NULL or duplicate values.

```
CREATE TABLE persons (
id INT NOT NULL PRIMARY KEY,
name VARCHAR(30) NOT NULL,
birth date DATE,
phone VARCHAR( I 5) NOT NULL
);
```


UNIQUE Constraint

The UNIQUE constraint restricts one or more columns to contain unique values within a table.

Although both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, use a UNIQUE constraint instead of a PRIMARY KEY constraint when you want to enforce the uniqueness of a column, or combination of columns, that is not the primary key.

The following SQL statement creates a table named persons and specifies the phone column as unique. That means this field does not allow duplicate values.

```
CREATE TABLE persons (
id INT NOT NULL PRIMARY KEY,
name VARCHAR(30) NOT NULL,
birth date DATE,
phone VARCHAR(1 5) NOT NULL UNIQUE
);
```

DEFAULT Constraint

The DEFAULT constraint specifies the default value for the columns.

A column default is some value that will be inserted in the column by the database engine when an INSERT statement doesn't explicitly assign a particular value.

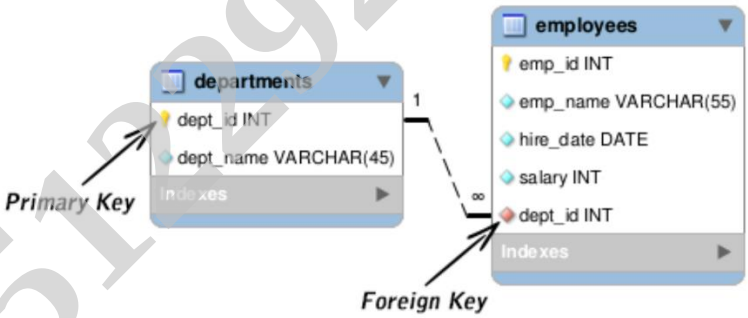
The following SQL statement creates a default for the country column.

```
CREATE TABLE persons (
id INT NOT NULL PRIMARY KEY,
name VARCHAR(30) NOT NULL,
birth date DATE,
phone VARCHAR(15) NOT NULL UNIQUE,
country VARCHAR(30) NOT NULL DEFAULT 'Australia');
```

FOREIGN KEY Constraint

A foreign key (FK) is a column or combination of columns that is used to establish and enforce a relationship between the data in two tables.

Here's a sample diagram showing the relationship between the employees and departments table. If you look at it carefully, you will notice that the dept id column of the employees table matches the primary key column of the departments table. Therefore, the dept id column of the employees table is the foreign key to the departments table.



In MySQL you can create a foreign key by defining a FOREIGN KEY constraint when you create a table as follow. The following statement establishes a foreign key on the dept_id column of the employees table that references the dept_id column of the departments table.

```
CREATE TABLE employees (
emp_id INT NOT NULL PRIMARY KEY,
cmp_name VARCHAR(55) NOT NULL,
hire date DATE NOT NULL,
salary INT,
dept id INT,
FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

CHECK Constraint

The CHECK constraint is used to restrict the values that can be placed in a column. For example, the range of values for a salary column can be limited by creating a CHECK constraint that allows values only from 3,000 to 10,000. This prevents salaries from being entered beyond the regular salary range. Here's an example.

```
CREATE TABLE employees (
emp _id INT NOT NULL PRIMARY KEY,
emp_name VARCHAR(55) NOT NULL,
hire date DATE NOT NULL,
salary INT NOT NULL CHECK (salary >- 3000 AND
```

```
salary <- 10000),
dept id INT,
FOREIGN KEY (dept_id) REFERENCES
departments(dept_id)
);
```

SQL DROP TABLE

A SQL DROP TABLE statement is used to delete a table definition and all data from a table. This is very important to know that once a table is deleted all the information available in the table is lost forever, so we have to be very careful when using this command.

Let's see the syntax to drop the table from the database.

```
DROP TABLE "table name";
```

SQL RENAME TABLE

SQL RENAME TABLE syntax is used to change the name of a table. Sometimes, we choose non-meaningful name for the table. So, it is required to be changed.

Let's see the syntax to rename a table from the database.

```
ALTER TABLE table_name
RENAME TO new _table name;
EX.
ALTER TABLE STUDENTS
RENAME TO ARTISTS;
```

SQL ALTER TABLE

The ALTER TABLE statement is used to add, modify, or delete columns in an existing table. It is also used to rename a table. You can also use SQL ALTER TABLE command to add and drop various constraints on an existing table.

SQL ALTER TABLE Add Column

If you want to add columns in SQL table, the SQL alter table syntax is given below:

```
ALTER TABLE table name ADD column_name column-
Definition:
```

If you want to add multiple columns in table, the SQL table will be.

```
ALTER TABLE table name
ADD (column_1 column-definition,
column_2 column-definition,
```

```
…….
column_n column-definition);
```

SQL ALTER TABLE Modify Column

If you want to modify an existing column in SQL table, syntax is given below.

```
ALTER TABLE table_name MODIFY column_name
column_ type;
```

If you want to modify multiple columns in table, the SQL table will be.

```
ALTER TABLE table name
MODIFY (column_1 column_type,
column_2 column_type,
….
column_n column_type);
```

SQL ALTER TABLE DROP Column

The syntax of alter table drop column is given below:

```
ALTER TABLE table name DROP COLUMN
column_name;
```

SQL ALTER TABLE RENAME Column

The syntax of alter table rename column is given below:

```
ALTER TABLE table name

RENAME COLUMN old_name to new_name;
```

SQL Insert Records

The INSERT INTO statement is used to insert new records in a table. It is possible to write the INSERT INTO statement in two ways. The first way specifies both the column names and the values to be inserted.

```
INSERT INTO table_name (columnl, column2,
column3, ..) VALUES (valuel, valueZ, values, ...);
```

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table.

The INSERT INTO syntax would be as follows:

```
INSERT INTO table name VALUES (valuel, value2,
value3,….);
```

To add new record in following table

This "Persons" table:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger

```
INSERT INTO Persons VALUES (‘Hetland', ‘Camilla’,
'Hagabakka 24', 'Sandnes')
```

The result is.

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes

To add data to specific columns

```
INSERT INTO Persons (LastName, Address) VALUES
(Rasmussen’, 'Storgt 67")
```

The result is.

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen		Storgt 67	

UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table name
SET column = valuel, column2 = value2, ...
WHERE condition.
```

Note: Be careful when updating records in a tablet Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all the records in the table will be updated.

In the following Person Table

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen		Storgt 67	

To add the first name in the third record

```
UPDATE Person SET FirstName='Nina' WHERE
LastName='Rasmussen'
```

The result is.

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen	Nina	Storgt 67	

To add the city name and update the address in the third line.

```
UPDATE Person SET Address='Stien 12', City=Stavanger"
WHERE LastName='Rasmussen'
```

The result is.

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen	Nina	Stien 12	Stavanger

DELETE Statement

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

```
DELETE FROM fable name WHERE condition;
```

Person Table

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen	Nina	Stien 12	Stavanger

To delete record number 3

```
DELETE FROM Person WHERE LastName =
'Rasmussen'
```

The result is.

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes

To delete all records

```
DELETE FROM table name

Or

DELETE * FROM table name
```

Select Statement

Used to extract data from the table and write as follows.

```
SELECT column_name(s) FROM table name
```

For Example Person Table

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

want to display LastName, FirstName from this table

```
SELECT LastName FirstName FROM Persons
```

The result is.

LastName	FirstName
Hansen	Ola
Svendson	Tove
Pettersen	Kari

To display all data from the table, use the following statement.

```
SELECT * FROM Persons
```

The result is.

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

Distinct Statement

Add the word Distinct to the sentence enables you to display values without repeating.

Example: The following table represents the order table data that contains duplicate values.

Company	OrderNumber
Acer	8153
DELL	8515
HP	4136
DELL	5516

If we use the following statement.

```
SELECT Company FROM Orders
```

The result will be.

Company
Acer
DELL
HP
DELL

If we use distinct keyword

```
SELECT DISTINCT Company FROM Orders
```

The result will be.

Company
Acer
DELL
HP

Where condition

It comes after the Select statement and contains the required condition.

```
SELECT column FROM table WHERE column operator value.
```

List of operators can used with where condition.

OPERATOR	DESCRIPTION
=	equal
!= or <>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
Between	the condition is between two values
Like	to search for similar words
in	to search on specific values

Person Table

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980
Pettersen	Kari	Storgt 20	Stavanger	1960

```
SELECT * FROM Persons WHERE City='Sandnes'
```

The result is.

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980

Important Note: To use a quotation mark with a sentence (') you must use a single quotation mark with text values such as names. As for the digital data you write without marks.

Like

Used to determine the search by a specific word, for example search for all the names that contain a letter or a particular word. and write as follows.

```
SELECT column FROM table WHERE column LIKE pattern
```

The% sign is used to make up for any number of characters before and after the word or letter we are looking at Search for people whose first name starts with the letter S

```
SELECT * FROM Persons WHERE FirstName LIKE 'S%'
```

The result is.

LastName	FirstName	Address	City	Year
Svendson	Stale	Kaivn 18	Sandnes	1980

Search for the names of people whose tirst name ends with the letter e

```
SELECT * FROM Persons WHERE FirstName LIKE '%e'
```

The result is.

LastName	FirstName	Address	City	Year
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980

Search for people whose first name includes the letter O.

```
SELECT * FROM Persons WHERE FirstName LIKE
'%O%'
```

The result is.

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978

AND - OR

Are used to associate two or more conditions.

- AND displays the results if all conditions are met

- OR operator displays the results if any condition is met

For example, Person Table

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Svendson	Stale	Kaivn 18	Sandnes

Use the operator AND

```
SELECT * FROM Persons WHERE FirstName="T'ove' AND
LastName='Svendson'
```

The result is.

LastName	FirstName	Address	City
Svendson	Tove	Borgvn 23	Sandnes

Use the operator OR

```
SELECT * FR OM Persons WHERE firstname='Tove' OR
lastname='Svendson'
```

The result is.

LastName	FirstName	Address	City
Svendson	Tove	Borgvn 23	Sandnes
Svendson	Stale	Kaivn 18	Sandnes

Use OR — AND together.

```
SELECT * FROM Persons WHERE (FirstName="Tove' OR
FirstName=' Ola") AND LastName='Svendson'
```

The result is.

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Svendson	Stale	Kaivn 18	Sandnes

IN Operator

It has several uses including that you can specify the value to display if you are sure, it is in a field.

```
SELECT column_name FROM table name WHERE
column_name IN (valuel,value2,..).
```

Person Table

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Nordmann	Anna	Neset 18	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

```
SELECT * FROM Persons WHERE LastName IN
('Hansen','Pettersen')
```

The result is.

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

It can also be used to link more than one query together

```
SELECT * FROM table 1_name WHERE FirstName in
(SELECT * FROM table2_name)
```

In this example we display all values from the first table name _table 1, Provided that the FirstName field is in the second table name table 2.

BETWEEN ... AND Operator

Used to display a data set between two text, number, or date values.

```
SELECT column_name FROM table name WHERE
column name BETWEEN value 1 AND value 2.
```

Person Table

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980

To display data between 1950 and 1979

```
SELECT * FROM Persons WHERE Year BETWEEN 1950
AND 1979
```

The result is.

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978

To display data not between 1950 and 1979

```
SELECT * FROM Persons WHERE Year Not
BETWEEN1950 AND 1979
```

The result is.

LastName	FirstName	Address	City	Year
Svendson	Stale	Kaivn 18	Sandnes	1980

Order by Keyword

Used to sort the result of the query by a specific field.

Company	OrderNumber
DELL	5516
DELL	8515
HP	4136
Acer	8153

We will execute the following query to display the data sorted alphabetically by company name.

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company
```

The result is.

Company	OrderNumber
Acer	8153
DELL	8515
DELL	5516
HP	4136

Note that the data appears in order by Company field if we want to order more than one field as follows:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company, OrderNumber
```

The result is.

Company	OrderNumber
Acer	8153
DELL	5516
DELL	8515
HP	4136

What if we wanted to reverse the order of how this is done?

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company DESC
```

The result is.

Company	OrderNumber
HP	4136
DELL	8515
DELL	5516
Acer	8153

JOIN

This is used to link tables to extract useful information. Sometimes we need to display data from two or more tables. To make the result complete, you must create a

relationship. Linking between tables is done by Key Primary, the field in which the primary key cannot be duplicated. In the following table, the ID_Employee field is the primary key field of the Employee table. In the second table, the ID_Order field is the primary key field of the table and has an external (foreign) key which is a field (Employee ID).

First Table: Employees

Employee_ID	Name
1	Hansen, Ola
2	Svendson, Tove
3	Svendson, Stephen
4	Pettersen, Kari

Second Table: Orders

Prod_ID	Product	Employee_ID
234	Printer	01
657	Table	02
865	Chair	03

Example: We want to know who ordered a product and what the product is

```
SELECT Employees.Name, Orders.Product FROM
Employees, Orders
WHERE Employees. Employee ID=Orders. Employee ID
```

The result.

Name	Product
Hansen, Ola	Printer
Svendson, Tove	Table
Svendson, Stephen	Chair

Example: We want to know who ordered the product Printer.

```
SELECT Employees.Name FROM Employees, Orders
WHERE Employees. Employee ID=Orders.Employee ID
AND Orders.Product="Printer'
```

The result

Name
Hansen, Ola

JOIN INNER

Displays all related data between the two tables.

```
SELECT fieldl, field2, field3 FROM first table INNER
JOIN second table ON first table.keyfield =
second _table.foreign keyfield
```

For example

```
SELECT Employees.Name, Orders.Product FROM
Employees INNER JOIN Orders ON
Employees. Employee ID=Orders. Employee ID
```

The result

Name	Product
Hansen, Ola	Printer
Svendson, Tove	Table
Svendson, Stephen	Chair

LEFT JOIN

Displays all data from the first table Employee even if it is not in the second table.

```
SELECT fieldl, field2, field3 FROM first table LEFT JOIN second table ON first table.keyfield = second _table.foreign keyfield
```

For example

```
SELECT Employees.Name, Orders.Product FROM
Employees LEFT JOIN Orders ON
Employees. Employee ID=Orders. Employee ID
```

The result

Name	Product
Hansen, Ola	Printer
Svendson, Tove	Table
Svendson, Stephen	Chair
Pettersen, Kari	

RIGHT JOIN

Displays all data from the second table, even if it is not in the first table.

```
SELECT fieldl, field2, field3 FROM first table RIGHT
JOIN second table ON first table.keyfield =
second _table.foreign keyfield
```

For example

```
SELECT Employees.Name, Orders.Product FROM Employees RIGHT JOIN Orders ON Employees. Employee ID=Orders. Employee ID
```

The result is.

Name	Product
Hansen, Ola	Printer
Svendson, Tove	Table
Svendson, Stephen	Chair

Functions

SQL contains many functions, whether textual or arithmetic. Any function consists of the following.

```
SELECT function (column) FROM table.
```

Examples of Functions

Function	Description
AVG (column)	Finds the average of the selected field
COUNT (column)	Know the number of rows (records) in the field without empty records
COUNT (*)	Know the number of rows in the table
First (column)	Know the value of the first record in the field
last (column)	Know the value of the last record in the field
Max (column)	Know the largest value in the field
Min (column)	Know the smallest value of a field
SUM (column)	Know the total values in the field

DB Exercise 1

QI) Discuss the difference between Primary Key and foreign Key (Give example to illustrate your answer)

Q2) Draw the correct relation between Teacher & student in Faculty Database System.

Q3) Give example & draw the relation in Hospital Database as:

A) One — One

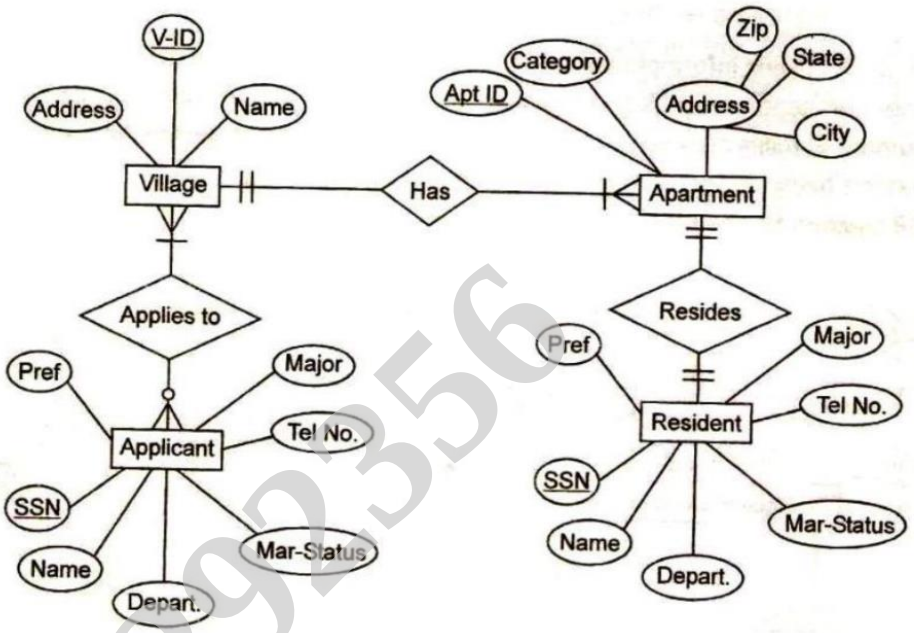
B) One — Many

C) Many - Many

DB Exercise 2

Q1) raw ERD for the following case study: A Country bus Company owns several buses. Each bus is allocated to a particular route, although some routes may have several buses. Each route passes through several towns. One or more drivers are allocated to each stage of a route, which corresponds to a journey through some or all the towns on a route. Some of the towns have a garage where buses are kept and each of the buses are identified by the registration number and can carry different numbers of passengers, since the vehicles vary in size and can be single or double-decked. Each route is identified by a route number and information is available on the average number of passengers carried per day for each route. Drivers have an employee number, name, address, and sometimes a telephone number.

Q2) Map the following ERD.



References

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Database system concepts.

D. Bălăceanu, "Components of a Business Intelligence software solution," Inform. Econ., vol. 2, no. 2, pp. 67–73, 2007.

J. Lloyd, "Identifying Key Components of Business Intelligence Systems and Their Role in Managerial Decision making," vol. 1277, no. February 2011, p. 76, 2011.

J. Ranjan, "Business Intelligence: Concepts, Components, Techniques and Benefits," J. Theor. Appl. Inf. Technol., vol. 9, p. 60, 2009.

Dennis, A. & Wixom, B. & Both, R. (2012). *Systems Analysis & Design*. Wiley.

Gowthami, K., & Kumar, M. P. (2017). Study on business intelligence tools for enterprise dashboard development. *International Research Journal of Engineering and Technology*, *4*(4), 2987-2992.

Grossmann, W., & Rinderle-Ma, S. (2015). Fundamentals of business intelligence.

Alpar, P., & Schulz, M. (2016). Self-service business intelligence. *Business & Information Systems Engineering*, *58*, 151-155.

Rifaie, M., Kianmehr, K., Alhajj, R., & Ridley, M. J. (2008, July). Data warehouse architecture and design. In *2008 IEEE International Conference on Information Reuse and Integration* (pp. 58-63). IEEE.

Lee, "Big data: Dimensions, evolution, impacts, and challenges," Bus. Horiz., vol. 60, no. 2017, pp. 293–303, 2017.

Elmasri, R. (2016). *Fundamentals of Database Systems*. Pearson.

Ponniah, P. (2011). Data warehousing fundamentals for IT professionals. John Wiley & Sons.

Paul, P., Aithal, P. S., & Bhimali, A. (2018). Business informatics: with special reference to big data as an emerging area: a basic review. International Journal on Recent Researches in Science, Engineering & Technology (IJRRSET), 6(4), 21-29.

Biography

Dr. Mohamed Abdel Salam, Mohamed Abdelsalam is currently an Associate Professor in the Information Systems Department, Faculty of Commerce & Business Administration, Helwan University, Cairo, Egypt. He earned his master's degree in information technology from the Faculty of Computers & Information, Helwan University, Cairo, Egypt, in 2013. His master's project focused on a proposed framework for applying Business Intelligence to enhance Ontology learning styles. He received his Ph.D. in information systems in July 2018, specializing in Constructing a Virtual Reality E-learning Environment Framework Based on Personalized Learning. He has over 10 years of teaching experience in the field of information systems and has published several research papers in various areas of information systems. His research interests encompass Data Analytics, Data Mining, Machine Learning, E-Learning solutions, Data Science, Data Warehousing, and Blockchain.

Dr. Menna Ibrahim Gabr, Assistant professor, Department of Business Information Systems (BIS) - Faculty of Commerce and Business Administration- Helwan University. Received Diploma and master's degree from faculty of Computer and Artificial Intelligence – Helwan University 2013, 2017 respectively. Received Ph.D. degree from faculty of Commerce and Business Administration- Helwan University 2023. The research interests include Data Science, Data Quality, Machine Learning, and Data Mining.