INDEX

SR.NO	**TOPIC**	**PAGE NO.**	
		From	**To**
1	OVERVIEW	3	9
2	ENTITY RELATION MODEL	10	21
3	RELATIONAL STRUCTURE	22	26
4	SCHEMA REFINEMENT AND NORMAL FORMS	27	31

Syllabus

UNIT I: Relational Model (15 Lectures)
(a) Overview: Overview of database management system , limitations of data processing environment, database approach, data independence, three level of abstraction, DBMS structure .
(b) Entity Relation Model: Entity, attributes, keys, relations, cardinality, participation, weak entities, ER diagram, Generalization, Specialization and aggregation, conceptual design with ER model, entity versus attribute, entity versus relationship, binary versus ternary relationship, aggregate versus ternary relationship.
(c) Relational Structure: Introduction to relational model, integrity constraints over relations.
(d) Schema refinement and Normal forms: Functional dependencies, first, second, third, and BCNF normal forms based on primary keys, lossless join decomposition.

CHAPTER 1: OVERVIEW

Topic Covered:

- **Overview:** Overview of database management system, limitations of data processing environment, database approach, data independence, three level of abstraction, DBMS structure.

Q. Write a short note On Database Management Systems?

- A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining, constructing,* and *manipulating* databases for various applications. **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.
- It is not necessary to use general-purpose DBMS software to implement a computerized database. We could write our own set of programs to create and maintain the database, in effect creating our own *special-purpose* DBMS software. In either case—whether we use a general-purpose DBMS or not—we usually have to employ a considerable amount of software to manipulate the database. We will call the database and DBMS software together a **database system.** Figure 01.01 illustrates these ideas.
- 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 technique form the focus of this book. This chapter briefly introduces the principles of database systems.

Q. Explain limitations of data processing environment?

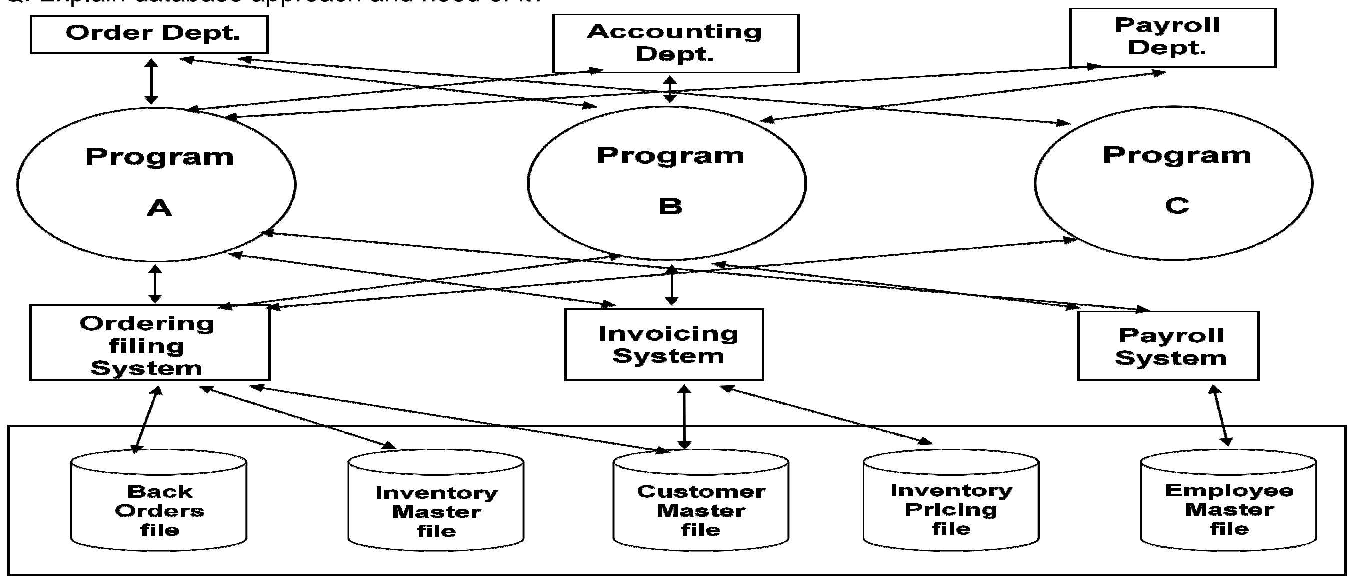
Keeping organizational information in a file-processing system has a number of major disadvantages:

- **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. 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 customer address may be reflected in savings-account records but not elsewhere in the system.
- **Difficulty in accessing data**. Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system 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 officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist.

Again, the officer 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 and efficient 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**. For example, the balance of a bank account may never fall below a prescribed amount (say, $25). 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 mechanical or electrical 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 $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was not credited to account *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. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. 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. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account 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 $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain either $450 or $400, rather than the correct value of $350. 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.
- **Security problems**. Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the 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 bank enterprise as a running example of a typical data-processing application found in a corporation.

Q. Explain database approach and need of it?



Why the Database Approach?

- Application needs *constantly changing*
- Ad hoc questions need *rapid answers*
- Need to reduce *long lead times* and *high cost* in new application development
- Lots of *data shared* throughout the organization
- Need to *improve data consistency* and *control access* to data
- Substantial dedicated programming assistance typically not available

Advantages of Using the Database Approach

- More information from given data
- Ad hoc queries can be performed
- Redundancy can be reduced
- Inconsistency can be avoided
- Security restriction can be applied
- Data independence
  - more cost-effective: reduced development time, flexibility, economies of scale
- Controlling redundancy in data storage and in development and maintenance.
- Sharing of data among multiple users.
- Providing persistent storage for program objects (in Object-oriented DBMS's – see Chs. 20-22)
- Providing storage structures for efficient query processing
- Providing backup and recovery services.
- Providing multiple interfaces to different classes of users.
- Representing complex relationships among data.
- Enforcing integrity constraints on the database.
- Drawing Inferences and Actions using rules.

Q. Explain Data Independence?

- **Data Independence**

  The three-schema architecture can be used to explain the concept of **data independence,** which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 01.04(a) should not be affected by changing the GRADE_REPORT file shown in Figure 01.02 into the one shown in Figure 01.05(a). Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval of SECTION records (Figure 01.02) by Semester and Year should not require a query such as "list all sections offered in fall 1998" to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

- Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence is accomplished because, when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.
- The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

Q. Explain three level of abstraction?

- **<u>Data Abstraction:</u>**
  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-systems users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' 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. 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 relationship 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. Most high-level programming languages
  support the notion of a record type. For example, in a Pascal-like language, we may declare a record as follows:
  type *customer* = **record**
  customer-id : string;
  customer-name : string;
  customer-street : string;
  customer-city : string;
  end;
- This code defines a new record type called *customer* with four fields. Each field has a name and a type associated with it. A banking enterprise may have several such record types, including
  - *account*, with fields *account-number* and *balance*
  - *employee*, with fields *employee-name* and *salary*
- At the physical level, a *customer*, *account*, or *employee* record can be described as a block of consecutive storage locations (for example, words or bytes). The language
  view 1 view 2
  logical
  level
  physical
  level
  … view n
  view level

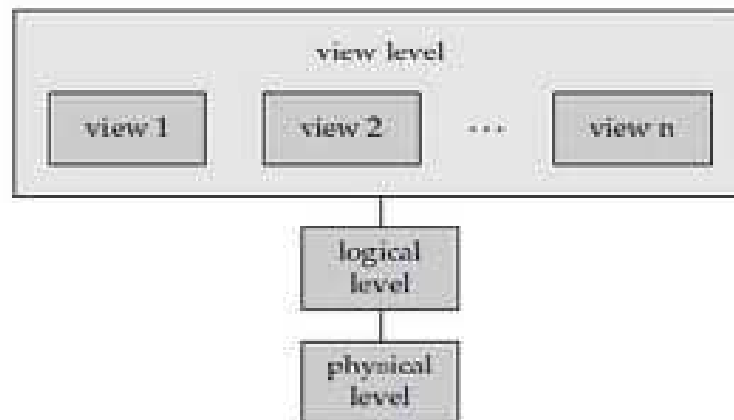
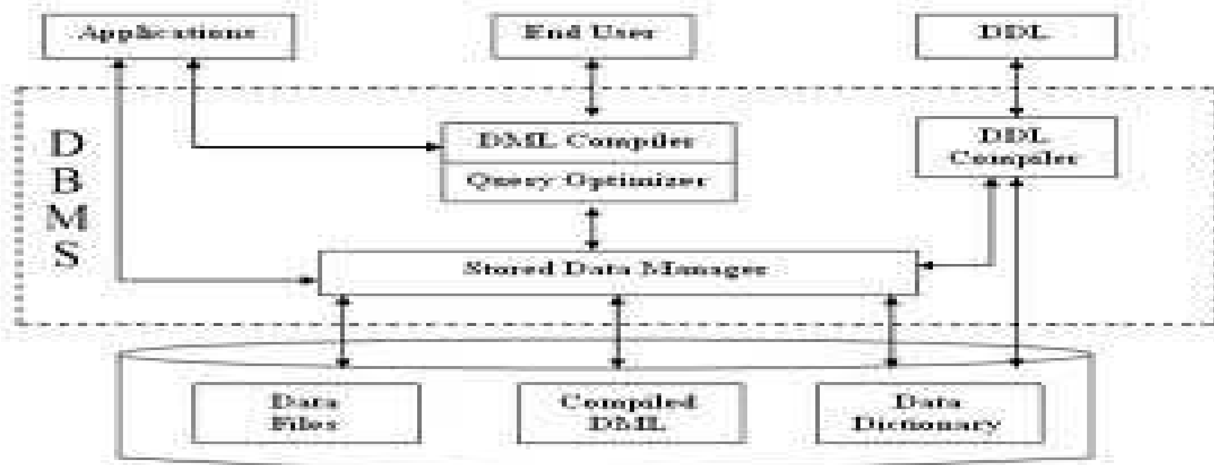


Figure 1.1 The three levels of data abstraction.

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 a set of application programs that hide details of the data types. Similarly, at the view level, several views of the database are defined, and database users see 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, tellers in a bank see only that part of the database that has information on customer accounts; they cannot access information about salaries of employees.

Q. Write a short note on DBMS structure?



The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a simplification: SQL commands can be embedded in host language application programs, e.g., Java or COBOL programs. We ignore these issues to concentrate on the core DBMS functionality.)
When a user issues a query, the parsed query is presented to a **query optimizer**, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An **execution plan** is a blueprint for evaluating a query, and is usually represented as a tree of relational operators (with annotations that contain additional detailed information about which access methods to use, etc.). We discuss query optimization in Chapter 13. Relational operators serve as the building blocks for evaluating queries posed against the data. The implementation of these operators .
The code that implements relational operators sits on top of the file and access methods layer. This layer includes a variety of software for supporting the concept of a **file**, which, in a DBMS, is a collection of pages or a collection of records. This layer typically supports a **heap file**, or _le of unordered pages, as well as indexes. In addition to keeping track of the pages in a file, this layer organizes the information within a page. File and page level storage issues are considered in Chapter 7. File organizations and indexes are considered in Chapter 8.
The files and access methods layer code sits on top of the **buffer manager**, which brings pages in from disk to main memory as needed in response to read requests. Buffer management is discussed in Chapter 7.
The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through (routines provided by) this layer, called the **disk space manager**. This layer is discussed in Chapter 7.
The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the **transaction manager**, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the **lock manager**, which keeps track of requests for locks and grants locks on database objects when they become available; and the **recovery manager**, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and _le and access method layers must interact with these components. We discuss concurrency control and recovery in detail in Chapter 18.

Quick Revision

* A database management system (DBMS) is software that supports management of large collections of data. A DBMS provides efficient data access, data independence, data integrity, security, quick application development, support for concurrent access, and recovery from system failures.
* Storing data in a DBMS versus storing it in operating system _les has many advantages. Using a DBMS provides the user with data independence, efficient data access, automatic data integrity, and security.
* The structure of the data is described in terms of a *data model* and the description is called a *schema*. The *relational model* is currently the most popular data model.
* A DBMS distinguishes between *external*, *conceptual*, and *physical schema* and thus allows a view of the data at three levels of abstraction. Physical and logical *data independence*, which are made possible by these three levels of abstraction, insulate the users of a DBMS from the way the data is structured and stored inside a DBMS.
* A *query language* and a *data manipulation language* enable high-level access and modification of the data.
* A *transaction* is a logical unit of access to a DBMS. The DBMS ensures that either all or none of a transaction's changes are applied to the database. For performance reasons, the DBMS processes multiple transactions concurrently, but ensures that the result is equivalent to running the transactions one after the other in some order. The DBMS maintains a record of all changes to the data in the *system log*, in order to undo partial transactions and recover from system crashes. *Check pointing* is a periodic operation that can reduce the time for recovery from a crash.
* DBMS code is organized into several modules: the disk space manager, the buffer manager, a layer that supports the abstractions of _les and index structures, a layer that implements relational operators, and a layer that optimizes queries and produces an execution plan in terms of relational operators.
* A *database administrator (DBA)* manages a DBMS for an enterprise. The DBA designs schemas, provide security, restores the system after a failure, and periodically tunes the database to meet changing user needs. Application programmers develop applications that use DBMS functionality to access and manipulate data, and end users invoke these applications.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Write a short note On Database Management Systems?	3
2	Q. Explain limitations of data processing environment?	3
3	Q. Explain database approach and need of it?	5
4	Q. Explain Data Independence?	5
5	Q. Explain three level of abstraction?	6
6	Q. Write a short note on DBMS structure?	8

CHAPTER2: ENTITY RELATION MODEL

Topic Covered:

- **Entity Relation Model:** Entity, attributes, keys, relations, cardinality, participation, weak entities, ER diagram, Generalization, Specialization and aggregation, conceptual design with ER model, entity versus attribute, entity versus relationship, binary versus ternary relationship, aggregate versus ternary relationship.

The entity-relationship *(ER) data model* allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. In this chapter, we introduce the ER model and discuss how its features allow us to model a wide range of data faithfully.
The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed, and precise, description that can be implemented in a DBMS. We begin with an overview of database design in Section 2.1 in order to motivate our discussion of the ER model. Within the larger context of the overall design process, the ER model is used in a phase called *conceptual database design.* We then introduce the ER model in Sections 2.2, 2.3, and 2.4. In Section 2.5, we discuss database design issues involving the ER model. We conclude with a brief discussion of conceptual database design for large enterprises. We note that many variations of ER diagrams are in use, and no widely accepted standards prevail. The presentation in this chapter is representative of the family of ER models and includes a selection of the most popular features.

Q. Explain term Entity, attributes, key relationship?

- An **entity** is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy depart- An **entity** is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy depart-ment. It is often useful to identify a collection of similar entities. Such a collection is called an **entity set**. Note that entity sets need not be disjoint; the collection of toy department employees and the collection of appliance department employees may both contain employee John Doe (who happens to work in both departments). We could also define an entity set called Employees that contains both the toy and appliance department employee sets.
- An entity is described using a set of **attributes**. All entities in a given entity set have the same attributes; this is essentially what we mean by *similar*. (This statement is an oversimplification, as we will see when we discuss inheritance hierarchies in Section 2.4.4, but it success for now and highlights the main idea.) Our choice of attributes reflects the level of detail at which we wish to represent information about entities.
- For example, the Employees entity set could use name, social security number (ssn), and parking lot (lot) as attributes. In this case we will store the name, social security number, and lot number for each employee. However, we will not store, say, an employee's address (or gender or age).
- For each attribute associated with an entity set, we must identify a **domain** of possible values. For example, the domain associated with the attribute *name* of Employees might be the set of 20-character strings.2 As another example, if the company rates employees on a scale of 1 to 10 and stores ratings in a field called *rating*, the associated domain consists of integers 1 through 10. Further, for each entity set, we choose a *key.*
- A **key** is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one **candidate** key; if so, we designate one of them as the **primary** key. For now we will assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set; that is, the set of attributes contains a key. We will revisit this point in Section 2.4.3.
- The Employees entity set with attributes *ssn, name,* and *lot* is shown in Figure 2.1.
- An entity set is represented by a rectangle, and an attribute is represented by an oval.
- Each attribute in the primary key is underlined. The domain information could be listed along with the attribute name, but we omit this to keep the figures compact. The key is *ssn.*

- A **relationship type** *R* among *n* entity types , , . . ., defines a set of associations—or a **relationship set**—among entities from these types. As for entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name R.* Mathematically, the relationship set *R* is a set of **relationship instances** , where each associates n individual entities (, , . . .,), and each entity in is a member of entity type , 1 1 *j* 1 *n.* Hence, a relationship type is a mathematical relation on , , . . ., , or alternatively it can be defined as a subset of the Cartesian product x x . . . x . Each of the entity types , , . . ., is said to **participate** in the relationship type *R,* and similarly each of the individual entities , , . . ., is said to participate in the relationship instance = (, , . . .,).
- Informally, each relationship instance in *R* is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance represents the fact that the entities participating in are related in some way in the corresponding miniworld situation. For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department the employee works for. Each relationship instance in the relationship set WORKS_FOR associates one employee entity and one department entity. Figure 03.09 illustrates this example, where each relationship instance is shown connected to the employee and department entities that participate in *.* In the miniworld represented by Figure 03.09, employees e1, e3, and e6 work for department d1; e2 and e4 work for d2; and e5 and e7 work for d3.
- In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box (see Figure 03.02).

Q. Write a short note On Cardinality?

- **Mapping Cardinalities**
  - **Mapping cardinalities**, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
    Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets. In this section, we shall concentrate on only binary relationship sets.
    For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:
  - **One to one**. An entity in *A* is associated with *at most* one entity in *B*, and an entity in *B* is associated with *at most* one entity in *A*. (See Figure 2.4a.)
  - **One to many**. An entity in *A* is associated with any number (zero or more) of entities in *B*. An entity in *B*, however, can be associated with *at most* one entity in *A*. (See Figure 2.4b.)
  - **Many to one**. An entity in *A* is associated with *at most* one entity in *B*. An entity in *B*,however, can be associated with any number (zero or more) of entities in *A*. (See Figure 2.5a.)
  - **Many to many**. An entity in *A* is associated with any number (zero or more) of entities in *B*, and an entity in *B* is associated with any number (zero or more) of entities in *A*. (See Figure 2.5b.)
  - The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling.
  - As an illustration, consider the *borrower* relationship set. If, in a particular bank, a loan can belong to only one customer, and a customer can have several loans, then the relationship set from *customer* to *loan* is one to many. If a loan can belong to several customers (as can loans taken jointly by several business partners), the relationship set is many to many. Figure 2.3 depicts this type of relationship.

The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. There are two types of participation constraints—total and partial—which we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in a WORKS_FOR relationship instance (Figure 03.09). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation,** meaning that every entity in "the total set" of employee entities must be related to a

department entity via WORKS_FOR. Total participation is also called **existence dependency.** In Figure 03.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial,** meaning that *some* or "part of the set of" employee entities are related to a department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 03.02).

3.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types.** In contrast, **regular entity types** that do have a key attribute are sometimes called **strong entity types.** Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with some of their attribute values. We call this other entity type the **identifying** or **owner entity type** (Note 9), and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type (Note 10). A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship, because a weak entity cannot be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (LicenseNumber) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 03.02). The attributes of DEPENDENT are Name (the first name of the dependent), BirthDate, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, BirthDate, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to **own** the dependent entities that are related to it.

A weak entity type normally has a **partial key,** which is the set of attributes that can uniquely identify weak entities that are *related to the same owner entity* (Note 11). In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key. In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 03.02). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a composite attribute with component attributes Name, BirthDate, Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the weak entity type representation if there are many attributes. If the weak entity participates independently in relationship types other than its identifying relationship type, then it should not be modeled as a complex attribute.

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we shall illustrate in Chapter 4.

2.5 Entity-Relationship Diagram

As we saw briefly in Section 1.4, an **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model. Such a diagram consists of the following major components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link attributes to entity sets and entity sets to relationship sets
- **Double ellipses**, which represent multivalued attributes
- **Dashed ellipses**, which denote derived attributes

• **Double lines**, which indicate total participation of an entity in a relationship set
• **Double rectangles**, which represent weak entity sets (described later, in Section 2.6.)
Consider the entity-relationship diagram in Figure 2.8, which consists of two entity sets, *customer* and *loan*, related through a binary relationship set *borrower*. The attributes associated with *customer* are *customer-id*, *customer-name*, *customer-street*, and *customer-city*. The attributes associated with *loan* are *loan-number* and *amount*. In Figure 2.8, attributes of an entity set that are members of the primary key are underlined. The relationship set *borrower* may be many-to-many, one-to-many, many-to-one, or one-to-one. To distinguish among these types, we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question.
• A directed line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a one-to-one or many-to-one relationship set, from *customer* to *loan*; *borrower* cannot be a many-to-many or a one-to-many relationship set from *customer* to *loan*.

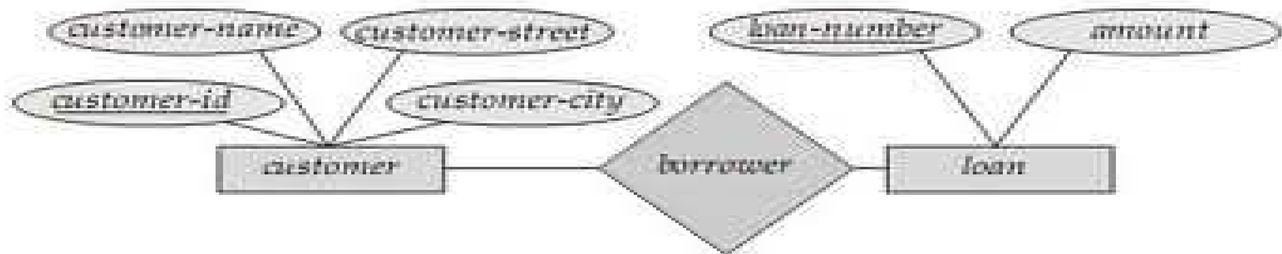
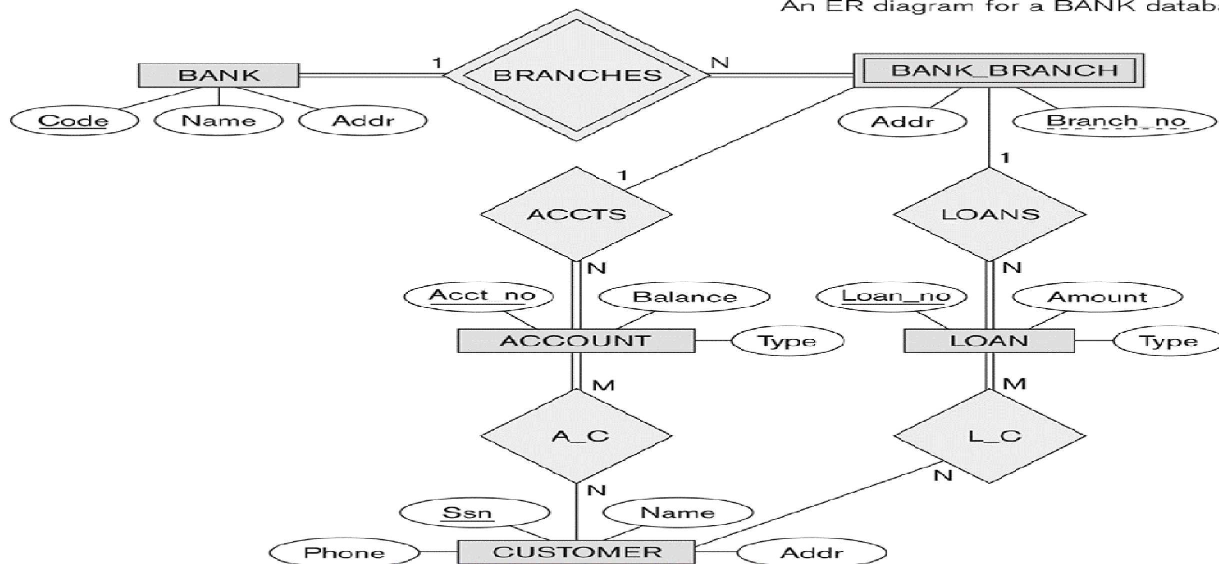


Figure E-R diagram corresponding to customers and loans.

An ER diagram for a BANK database schema.



• An undirected line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a many-to-many or one-to-many relationship set from *customer* to *loan*.
Returning to the E-R diagram of Figure 2.8, we see that the relationship set *borrower* is many-to-many. If the relationship set *borrower* were one-to-many, from *customer* to *loan*, then the line from *borrower* to *customer* would be directed, with an arrow pointing to the *customer* entity set (Figure 2.9a). Similarly, if the relationship set *borrower* were many-to-one from *customer* to *loan*, then the line from *borrower* to *loan* would have an arrow pointing to the *loan* entity set (Figure 2.9b). Finally, if the relationship set *borrower* were one-to-one, then both lines from *borrower* would have arrows:

Q. Explain Generalization, Specialization and aggregation?

- **Specialization** is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that form a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among EMPLOYEE entities based on the *job type* of each entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the *method of pay.*
- Figure 04.01 shows how we represent a specialization diagrammatically in an **EER diagram.** The subclasses that define a specialization are attached by lines to a circle, which is connected to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship (Note 7). Attributes that apply only to entities of a particular subclass—such as Typing Speed of SECRETARY—are attached to the rectangle representing that subclass. These are called **specific attributes** (or **local attributes**) of the subclass. Similarly, a subclass can participate in **specific relationship types,** such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO relationship in Figure 04.01. We will explain the **d** symbol in the circles of Figure 04.01 and additional EER diagram notation shortly.
- Figure 04.02 shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, e1 is shown in both EMPLOYEE and SECRETARY in Figure 04.02. As this figure suggests, a superclass/subclass relationship such as EMPLOYEE/SECRETARY somewhat resembles a 1:1 relationship *at the instance level* (see Figure 03.12). The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a super class/subclass relationship the entity in the subclass is the same real-world entity as the entity in the superclass but playing a *specialized role*—for example, an EMPLOYEE specialized in the role of SECRETARY, or an EMPLOYEE specialized in the role of TECHNICIAN.
- There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, the SECRETARY subclass may have an attribute Typing Speed, whereas the ENGINEER subclass may have an attribute Engineer Type, but SECRETARY and ENGINEER share their other attributes as members of the EMPLOYEE entity type.
- The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEEs can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Figure 04.01.
- In summary, the specialization process allows us to do the following:
- Define a set of subclasses of an entity type.
- Establish additional specific attributes with each subclass.
- Establish additional specific relationship types between each subclass and other entity types or other subclasses.
- **Generalization**
- We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses.** For example, consider the

entity types CAR and TRUCK shown in Figure 04.03(a); they can be generalized into the entity type VEHICLE, as shown in Figure 04.03(b). Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

- Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 04.03 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 04.01 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will *not* use this notation, because the decision as to which process is more appropriate in a particular situation is often subjective. Appendix A gives some of the suggested alternative diagrammatic notations for schema diagrams/class diagrams.
- So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type;* that is why superclasses and subclasses are shown in rectangles in EER diagrams (like entity types). We now discuss in more detail the properties of specializations and generalizations.
- **Aggregation** is an abstraction concept for building composite objects from their component objects. There are three cases where this concept can be related to the EER model. The first case is the situation where we aggregate attribute values of an object to form the whole object. The second case is when we represent an aggregation relationship as an ordinary relationship. The third case, which the EER model does not provide for explicitly, involves the possibility of combining objects that are related by a particular relationship instance into a *higher-level aggregate object.* This is sometimes useful when the higher-level aggregate object is itself to be related to another object. We call the relationship between the primitive objects and their aggregate object **IS-A-PART-OF;** the inverse is called **IS-A-COMPONENT-OF.** UML provides for all three types of aggregation.
- The abstraction of **association** is used to associate objects from several *independent classes.* Hence, it is somewhat similar to the second use of aggregation. It is represented in the EER model by relationship types and in UML by associations. This abstract relationship is called **IS-ASSOCIATED-WITH.**
- In order to understand the different uses of aggregation better, consider the ER schema shown in Figure 04.16(a), which stores information about interviews by job applicants to various companies. The class COMPANY is an aggregation of the attributes (or component objects) CName (company name) and CAddress (company address), whereas JOB_APPLICANT is an aggregate of Ssn, Name, Address, and Phone. The relationship attributes ContactName and ContactPhone represent the name and phone number of the person in the company who is responsible for the interview. Suppose that some interviews result in job offers, while others do not. We would like to treat INTERVIEW as a class to associate it with JOB_OFFER. The schema shown in Figure 04.16(b) is *incorrect* because it requires each interview relationship instance to have a job offer. The schema shown in Figure 04.16(c) is not allowed, because the ER model does not allow relationships among relationships (although UML does).
- One way to represent this situation is to create a higher-level aggregate class composed of COMPANY, JOB_APPLICANT, and INTERVIEW and to relate this class to JOB_OFFER, as shown in Figure 04.16(d). Although the EER model as described in this book does not have this facility, some semantic data models do allow it and call the resulting object a **composite** or **molecular object.** Other models treat entity types and relationship types uniformly and hence permit relationships among relationships (Figure 04.16c).
- To represent this situation correctly in the ER model as described here, we need to create a new weak entity type INTERVIEW, as shown in Figure 04.16(e), and relate it to JOB_OFFER. Hence, we can always represent these situations correctly in the ER model by creating additional entity types, although it may be conceptually more desirable to allow direct representation of

aggregation as in Figure 04.16(d) or to allow relationships among relationships as in Figure 04.16(c). The main structural distinction between aggregation and association is that, when an association instance is deleted, the participating objects may continue to exist. However, if we support the notion of an aggregate object—for example, a CAR that is made up of objects ENGINE, CHASSIS, and TIRES—then deleting the aggregate CAR object amounts to deleting all its component objects.

2.5 CONCEPTUAL DATABASE DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use
- binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.

Q. Explain Entity versus Attribute?

- While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it success to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has Address). This more complex alternative is necessary in two situations:
- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as \Find all employees with an address in Madison, WI."
- For another example of when to model a concept as an entity set rather than as an attribute, consider the relationship set (called Works In2) shown in Figure 2.14.

Entity vs. Attribute (Contd.)

- Works_In2 does not allow an employee to work in a department for two or more periods.
- Similar to the problem of wanting to record several addresses for an employee: we want to record *several values of the descriptive attributes for each instance of this relationship.*

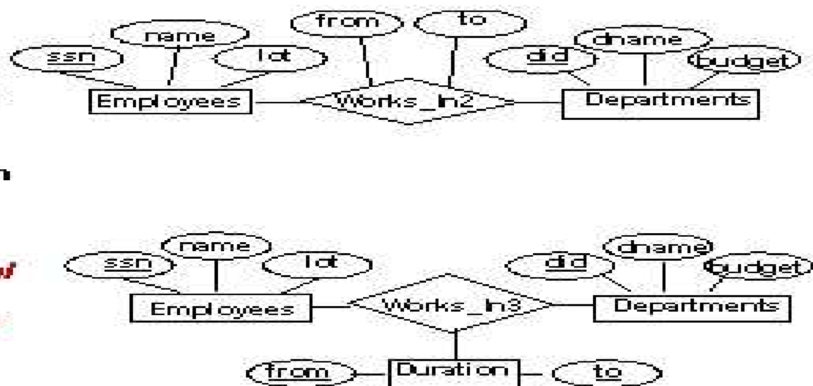


Figure 2.14 The Works In2 Relationship Set

- It differs from the Works In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.
- This possibility is ruled out by the ER diagram's semantics. The problem is that we want to record several values for the descriptive attributes for each instance of the Works In2 relationship. (This situation is analogous to wanting to record several addresses for each

employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.

- In some versions of the ER model, attributes are allowed to take on sets as values.
- Given this feature, we could make Duration an attribute of Works In, rather than an entity set; associated with each Works In relationship, we would have a set of intervals.
- This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

Q. Explain Entity versus relationship?

- **Entity versus Relationship**
- Consider the relationship set called Manages in Figure 2.6. Suppose that each depart- ment manager is given a discretionary budget (*dbudget*), as shown in Figure 2.16, in which we have also renamed the relationship set to Manages2.
- There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager-department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.
  - But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field. In general such redundancy could be
- significant and could cause a variety of problems. (We discuss redundancy and its attendant problems in Chapter 15.) Another problem with this design is that it is misleading.
- We can address these problems by associating *dbudget* with the appointment of the employee as manager of a *group* of departments. In this approach, we model the appointment as an entity set, say Mgr Appt, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field. In general such redundancy could be significant and could cause a variety of problems. (We discuss redundancy and its attendant problems in Chapter 15.)
- Another problem with this design is that it is misleading. We can address these problems by associating *dbudget* with the appointment of the employee as manager of a *group* of departments. In this approach, we model the appointment as an entity set, say Mgr Appt, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach is illustrated in Consider the ER diagram shown in Figure 2.18. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.
- Suppose that we have the following additional requirements: A policy cannot be owned jointly by two or more employees. Every policy must be owned by some employee.

- is illustrated in

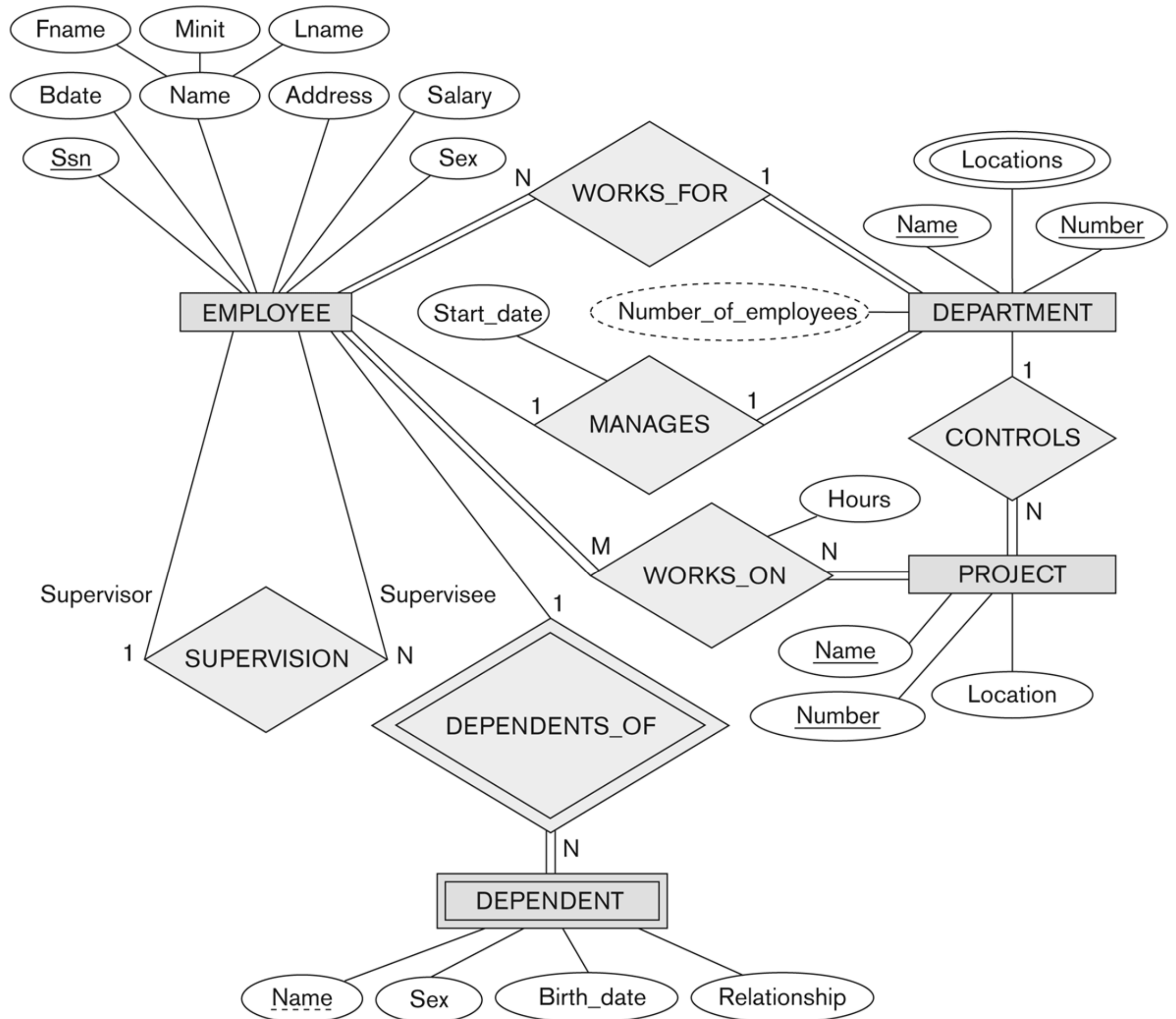


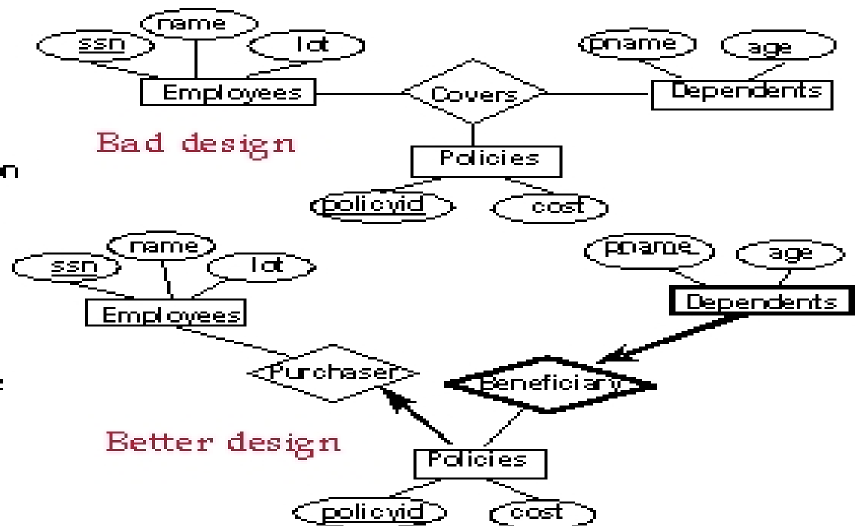
Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Q. Explain Binary versus ternary relationship?

- **Binary versus Ternary Relationships**
  Consider the ER diagram shown in Figure 2.18. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.
  Suppose that we have the following additional requirements:
  A policy cannot be owned jointly by two or more employees.
  Every policy must be owned by some employee.

Binary vs. Ternary Relationships

- **If each policy is owned by just 1 employee:**
  - Key constraint on Policies would mean policy can only cover 1 dependent!
- **What are the additional constraints in the 2nd diagram?**

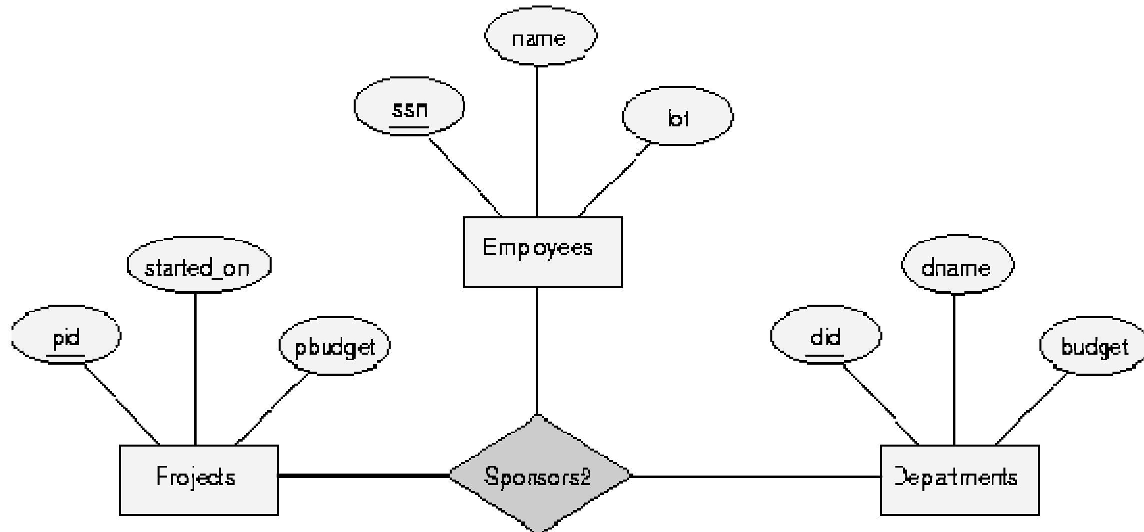


Dependents is a weak entity set, and each dependent entity is uniquely identified by taking *pname* in conjunction with the *policyid* of a policy entity (which, intuitively, covers the given dependent).
The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).
Even ignoring the third point above, the best way to model this situation is to use two binary relationships, as shown in Figure 2.19.
This example really had two relationships involving Policies, and our attempt to use a single ternary relationship (Figure 2.18) was inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figure 2.4 and also Figures 2.15 and 2.17.
As a good example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute *qty*) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier `can supply' certain parts, that a department `needs' some policy Revisited parts, or that a department `deals with' a certain supplier. No combination of these Relationships expresses the meaning of a contract adequately, for at least two reasons:
The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
We cannot represent the *qty* attribute of a contract cleanly.

Q. Explain aggregate versus ternary relationship?

- **Aggregation versus Ternary Relationships**
- As we noted in Section 2.4.5, the choice between using aggregation or a ternary relation- ship is mainly determined by the existence of a relationship that relates a *relationship set* to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in Figure 2.13. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees.
- If we don't need to record the *until* attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure 2.20.

- Consider the constraint that each sponsorship (of a project by a department) be mon- itored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship



Monitors in Figure 2.13. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.

Quick Revision

- Database design has six steps: requirements analysis, conceptual database design, logical database design, schema refinement, physical database design, and security design. Conceptual design should produce a high-level description of the data, and the entity-relationship (ER) data model provides a graphical approach to this design phase.
- In the ER model, a real-world object is represented as an *entity*. An *entity set* is a collection of structurally identical entities. Entities are described using *attributes*. Each entity set has a distinguished set of attributes called a *key* that can be used to uniquely identify each entity.
- A *relationship* is an association between two or more entities. A *relationship set* is a collection of relationships that relate entities from the same entity sets. A relationship can also have *descriptive attributes*.
- A *key constraint* between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at most one relationship. A *participation constraint* between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at least one relationship. The identity and existence of a *weak entity* depends on the identity and existence of another (*owner*) entity. *Class hierarchies* organize structurally similar entities through inheritance into sub- and superclasses.
- *Aggregation* conceptually transforms a relationship set into an entity set such that the resulting construct can be related to other entity sets. Development of an ER diagram involves important modeling decisions. A thorough understanding of the problem being modeled is necessary to decide whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. Conceptual design for large enterprises is especially challenging because data from many sources, managed by many groups, is involved.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain term Entity, attributes, key relationship?	10
2	Q. Write a short note On Cardinality?	11
3	Q. Explain Generalization, Specialization and aggregation?	14
4	Q. Explain Entity versus Attribute?	16
5	Q. Explain Entity versus relationship?	17
6	Q. Explain binary versus ternary relationship?	18
7	Q. Explain aggregate versus ternary relationship?	19

CHAPTER 3: RELATIONAL STRUCTURE

Topic Covered:

- Introduction to relational model, integrity constraints over relations.

3.1 INTRODUCTION TO THE RELATIONAL MODEL

Q. Explain relational model?

- The main construct for representing data in the relational model is a **relation**. A relation consists of a **relation schema** and a **relation instance**. The relation instance is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance. The schema specifies the relation's name, the name of each **field** (or **column**, or **attribute**), and the **domain** of each field. A domain is referred to in a relation schema by the **domain name** and has a set of associated **values**.
- We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema: Students(*sid:* string, *name:* string, *login:* string, *age:* integer, *gpa:* real)
- This says, for instance, that the field named *sid* has a domain named string. The set of values associated with domain string is the set of all character strings.
- We now turn to the instances of a relation. An **instance** of a relation is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a *table* in which each tuple is a *row*, and all rows have the same number of fields. (The term *relation instance* is often abbreviated to just *relation*, when there is no confusion with other aspects of a relation such as its schema.)
- An instance of the Students relation appears in Figure 3.1. The instance *S*1 contains

Figure 3.1 An Instance *S*1 of the Students Relation

- six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model| each relation is defined to be a *set* of unique tuples or rows.1 The order in which the rows are listed is not important. Figure 3.2 shows the same relation instance. If the fields are named, as in 1In practice, commercial systems allow tables to have duplicate rows, but we will assume that a relation is indeed a set of tuples unless otherwise noted.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53666	Jones	jones@cs	18	3.4
50000	Dave	dave@cs	19	3.3

Figure 3.2 An Alternative Representation of Instance *S*1 of Students

our schema definitions and figures depicting relation instances, the order of fields does

not matter either. However, an alternative convention is to list fields in a specific order and to refer to a field by its position. Thus *sid* is field 1 of Students, *login* is field 3, and so on. If this convention is used, the order of fields is significant. Most database systems use a combination of these conventions. For example, in SQL the named fields convention is used in statements that retrieve tuples, and the ordered fields convention is commonly used when inserting tuples.

- A relation schema specifies the domain of each field or column in the relation instance.
- A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database. For example, in Chapter 1, we discussed a university database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets In. An **instance** of a relational database is a collection of relation instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

These **domain constraints** in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the *type* of that field, in programming language terms, and restricts the values that can appear in the field.
More formally, let R(*f*1:D1, *: : :*, *fn*:Dn) be a relation schema, and for each *fi*, 1 *fi i fi n*,
let *Domi* be the set of values associated with the domain named Di. An instance of R
that satisfies the domain constraints in the schema is a set of tuples with *n* fields:
*f hf*1 : *d*1*; : : : ; fn* : *dni j d*1*2Dom*1*; : : : ; dn 2 Domn g*
The angular brackets *h: : :i* identify the fields of a tuple. Using this notation, the first Students tuple shown in Figure 3.1 is written as *hsid:* 50000, *name:* Dave, *login:*
dave@cs, *age:* 19, *gpa:* 3.3*i*. The curly brackets *f: : :g* denote a set (of tuples, in this definition). The vertical bar *j* should be read `such that,' the symbol *2* should be read `in,' and the expression to the right of the vertical bar is a condition that must be satisfied by the field values of each tuple in the set. Thus, an instance of R is defined as a set of tuples. The fields of each tuple must correspond to the fields in the relation schema.
Domain constraints are so fundamental in the relational model that we will henceforth consider only relation instances that satisfy them; therefore, *relation instance* means *relation instance that satisfies the domain constraints in the relation schema.*
The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality** of a relation instance is the number of tuples in it. In Figure 3.1, the degree of the relation (the number of columns) is five, and the cardinality of this instance is six.

3.2 INTEGRITY CONSTRAINTS OVER RELATIONS

A database is only as good as the information stored in it, and a DBMS must therefore help prevent the entry of incorrect information. An **integrity constraint (IC)** is a condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance. A DBMS **enforces** integrity constraints, in that it permits only legal instances to be stored in the database.
Integrity constraints are specified and enforced at different times:
1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. (In some situations, rather than disallow the change, the DBMS might instead make some compensating changes to the data to ensure that the database instance satisfies all ICs. In any case, changes to the database are not allowed to create an instance that violates any IC.)
Many kinds of integrity constraints can be specified in the relational model. We have already seen one example of an integrity constraint in the *domain constraints* associated with a relation schema (Section 3.1). In general, other kinds of constraints can be specified as well; for example, no two students have the same *sid* value. In this section
we discuss the integrity constraints, other than domain constraints, that a DBA or user can specify in the relational model.

Q. Write a short note on Key Constraints?

- **Key Constraints**
  - Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a key constraint. A **key constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple.
  - A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just *key*. In the case of the Students relation, the (set of fields containing just the) *sid* field is a candidate key.

  Let us take a closer look at the above definition of a (candidate) key. There are two parts to the definition:3
  - Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.
  - No subset of the set of fields in a key is a unique identifier for a tuple. 3The term *key* is rather overworked. In the context of access methods, we speak of *search keys*, which are quite different.
  - The first part of the definition means that in *any* legal instance, the values in the key fields uniquely identify a tuple in the instance. When specifying a key constraint, the DBA or user must be sure that this constraint will not prevent them from storing a `correct' set of tuples. (A similar comment applies to the specification of other kinds of ICs as well.) The notion of `correctness' here depends upon the nature of the data being stored. For example, several students may have the same name, although each student has a unique student id. If the *name* field is declared to be a key, the DBMS will not allow the Students relation to contain two tuples describing different students with the same name!
  - The second part of the definition means, for example, that the set of fields *fsid, nameg* is not a key for Students, because this set properly contains the key *fsidg*. The set *fsid, nameg* is an example of a **superkey**, which is a set of fields that contains a key.
  - Look again at the instance of the Students relation in Figure 3.1. Observe that two different rows always have different *sid* values; *sid* is a key and uniquely identifies a tuple. However, this does not hold for nonkey fields. For example, the relation contains two rows with *Smith* in the *name* field.
  - Note that every relation is guaranteed to have a key. Since a relation is a set of tuples, the set of *all* fields is always a superkey. If other constraints hold, some subset of the fields may form a key, but if not, the set of all fields is a key.
  - A relation may have several candidate keys. For example, the *login* and *age* fields of the Students relation may, taken together, also identify students uniquely. That is, *flogin, ageg* is also a key. It may seem that *login* is a key, since no two rows in the example instance have the same *login* value. However, the key must identify tuples uniquely in all possible legal instances of the relation. By stating that *flogin, ageg* is a key, the user is declaring that two students may have the same login or age, but not both. Out of all the available candidate keys, a database designer can identify a **primary** key. Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its *sid* value. As a consequence of referring to student tuples in this manner, tuples are frequently accessed by specifying their *sid* value. In principle, we can use any key, not just the primary key, to refer to a tuple. However, using the primary key is preferable because it is what the DBMS expects this is the significance of designating a particular candidate key as a primary key |and optimizes for. For example, the DBMS may create an index with the primary key fields as the search key, to make the retrieval of a tuple given its primary key value efficient. The idea of referring to a tuple is developed further in the next section.

Q. Write a short note on Foreign Key constraints?

- **Foreign Key Constraints**
  - Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common IC involving two relations is a *foreign key* constraint.
  - Suppose that in addition to Students, we have a second relation:

- Enrolled(*sid:* string, *cid:* string, *grade:* string)
- To ensure that only bona _de students can enroll in courses, any value that appears in the *sid* field of an instance of the Enrolled relation should also appear in the *sid* field of some tuple in the Students relation. The *sid* field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.
- This constraint is illustrated in Figure 3.4. As the figure shows, there may well be some students who are not referenced from Enrolled (e.g., the student with *sid=50000*).
- However, every *sid* value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

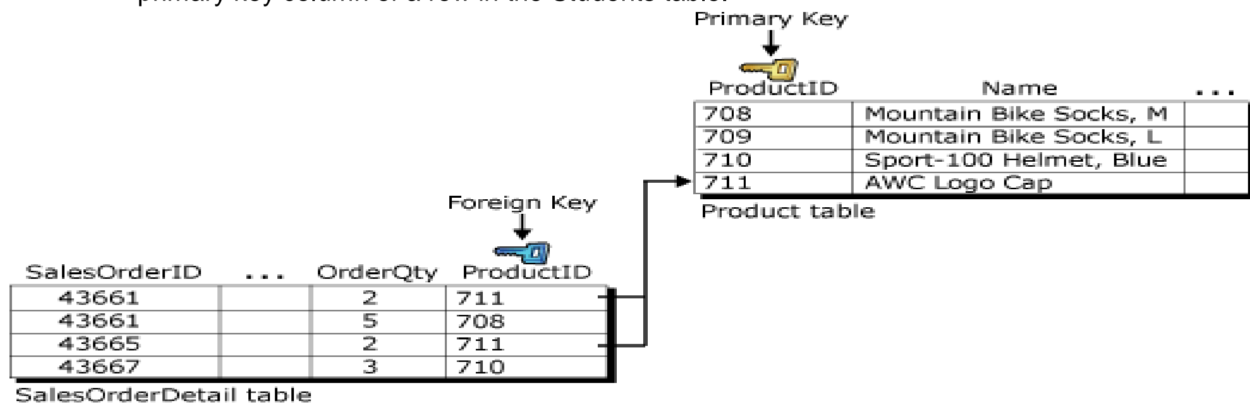


Figure 3.4 Referential Integrity

- If we try to insert the tuple *h55555, Art104, Ai* into *E*1, the IC is violated because there is no tuple in *S*1 with the id 55555; the database system should reject such an insertion. Similarly, if we delete the tuple *h53666, Jones, jones@cs, 18, 3.4i* from *S*1, we violate the foreign key constraint because the tuple *h53666, History105, Bi* in *E*1 contains *sid* value 53666, the *sid* of the deleted Students tuple. The DBMS
- should disallow the deletion or, perhaps, also delete the Enrolled tuple that refers to the deleted Students tuple. We discuss foreign key constraints and their impact on updates in Section 3.3.
- Finally, we note that a foreign key could refer to the same relation. For example, we could extend the Students relation with a column called *partner* and declare this column to be a foreign key referring to Students. Intuitively, every student could then have a partner, and the *partner* field contains the partner's *sid*. The observant reader will no doubt ask, \What if a student does not (yet) have a partner?" This situation is handled in SQL by using a special value called **null**. The use of *null* in a field of a tuple means that value in that field is either unknown or not applicable (e.g., we do not know the partner yet, or there is no partner). The appearance of *null* in a foreign key field does not violate the foreign key constraint. However, *null* values are not allowed to appear in a primary key field (because the primary key fields are used to identify a tuple uniquely). We will discuss *null* values further in Chapter 5.

Q. Write a short note on general constraints?

➢ **General Constraints**

- Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.
- For example, we may require that student ages be within a certain range of values; given such an IC specification, the DBMS will reject inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, the instance of Students shown in Figure 3.1 is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in Figure 3.5.

  Sid *name* *login* *age* *gpa*

53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.5 An Instance *S*2 of the Students Relation

- The IC that students must be older than 16 can be thought of as an extended domain constraint, since we are essentially defining the set of permissible *age* values more stringently than is possible by simply using a standard domain such as integer. In general, however, constraints that go well beyond domain, key, or foreign key constraints can be specified. For example, we could require that every student whose age is greater than 18 must have a gpa greater than 3.
- Current relational database systems support such general constraints in the form of *table constraints* and *assertions*. Table constraints are associated with a single table and are checked whenever that table is modified. In contrast, assertions involve several tables and are checked whenever any of these tables is modified. Both table constraints and assertions can use the full power of SQL queries to specify the desired restriction.
- We discuss SQL support for *table constraints* and *assertions* in Section 5.11 because a full appreciation of their power requires a good grasp of SQL's query capabilities.

Quick Revision

- The main element of the relational model is a *relation*. A *relation schema* describes the structure of a relation by specifying the relation name and the names of each field. In addition, the relation schema includes *domain constraints*, which are type restrictions on the fields of the relation.
- The number of fields is called the *degree* of the relation. The *relation instance* is an actual table that contains a set of *tuples* that adhere to the relation schema. The number of tuples is called the *cardinality* of the relation.
- SQL-92 is a standard language for interacting with a DBMS. Its *data definition language (DDL)* enables the creation (CREATE TABLE) and modification (DELETE, UPDATE) of relations.
- *Integrity constraints* are conditions on a database schema that every legal database instance has to satisfy. Besides domain constraints, other important types of ICs are *key constraints* (a minimal set of fields that uniquely identify a tuple) and *foreign key constraints* (fields in one relation that refer to fields in another relation). SQL-92 supports the specification of the above kinds of ICs, as well as more general constraints called *table constraints* and *assertions*.
- A *relational database query* is a question about the data. SQL supports a very expressive query language. There are standard translations of ER model constructs into SQL.
- Entity sets are mapped into relations. Relationship sets without constraints are also mapped into relations. When translating relationship sets with constraints, weak entity sets, class hierarchies, and aggregation, the mapping is more complicated.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain relational model?	22
2	Q. Write a short note on Key Constraints?	24
3	Q. Write a short note on Foreign Key constraints?	24
4	Q. Write a short note on general constraints?	25

CHAPTER4: SCHEMA REFINEMENT AND NORMAL FORMS

Topic Covered:

- Functional dependencies, first, second, third, and BCNF normal forms based on primary keys, lossless join decomposition.

Q. Explain functional dependencies?

- **FUNCTIONAL DEPENDENCIES**
  - A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*. Let *R* be a relation schema and let *X* and *Y* be nonempty sets of attributes in *R*. We say that an instance *r* of *R* satisfies the FD *X ! Y* 1 if the following holds for every pair of tuples *t*1 and *t*2 in *r*: If *t*1*:X* = *t*2*:X*, then *t*1*:Y* = *t*2*:Y* .
  - We use the notation *t*1*:X* to refer to the projection of tuple *t*1 onto the attributes in *X*, in a natural extension of our TRC notation (see Chapter 4) *t:a* for referring to attribute *a* of tuple *t*. An FD *X!Y* essentially says that if two tuples agree on the values in attributes *X*, they must also agree on the values in attributes *Y*.
  - Figure 15.3 illustrates the meaning of the FD *AB ! C* by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, *AB* is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the *A* field or the *B* field, they can differ in the *C* field without violating the FD. On the other hand, if we add a tuple *h*a1*; b*1*; c*2*; d*1*i* to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in the figure with the new tuple.

A	*B*	*C*	*D*
a1	b1	c1	d1
a1	*b1*	*c1*	*d2*
a1	*b2*	*c2*	*d1*
a2	*b1*	*c3*	*d1*

Figure 15.3 An Instance that Satisfies *AB ! C*

  - Recall that a *legal* instance of a relation must satisfy all specified ICs, including all specified FDs. As noted in Section 3.2, ICs must be identified and specified based on the semantics of the real-world enterprise being modeled. By looking at an instance of a relation, we might be able to tell that a certain FD does *not* hold. However, we can never deduce that an FD *does* hold by looking at one or more instances of the relation because an FD, like other ICs, is a statement about *all* possible legal instances of the relation. 1*X ! Y* is read as *X functionally determines Y*, or simply as *X determines Y*.
  - A primary key constraint is a special case of an FD. The attributes in the key play the role of *X*, and the set of all attributes in the relation plays the role of *Y*. Note, however, that the definition of an FD does not require that the set *X* be minimal; the additional minimality condition must be met for *X* to be a key. If *X ! Y* holds, where *Y* is the set of all attributes, and there is some subset *V* of *X* such that *V ! Y* holds, then *X* is a *superkey*; if *V* is a strict subset of *X*, then *X* is not a key.
  - In the rest of this chapter, we will see several examples of FDs that are not key constraints.

NORMAL FORMS

- Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several **normal forms** have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. The normal forms based on FDs are *_rst normal form (1NF)*, *second normal form*
- *(2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*. These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in **first normal form** if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our definition of the relational model. Although some of the newer database systems are relaxing this requirement,

in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

While studying normal forms, it is important to appreciate the role played by FDs. Consider a relation schema *R* with attributes *ABC*. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for redundancy. On the other hand, suppose that we have the FD *A ! B*. Now if several tuples have the same *A* value, they must also have the same *B* value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well. We will primarily discuss redundancy that is revealed by FD information. In Section 15.8, we discuss more sophisticated ICs called *multivalued dependencies* and *join dependencies* and normal forms based on them.

Q. write a short note on Boyce-Codd Normal Form?

- **Boyce-Codd Normal Form**
- Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and let *A* be an attribute of *R*. *R* is in **Boyce-Codd normal form** if for every FD *X ! A* that holds over *R*, one of the following statements is true:
  A 2 X; that is, it is a trivial FD, or *X* is a superkey.
- Note that if we are given a set *F* of FDs, according to this definition, we must consider each dependency *X ! A* in the closure F_+ to determine whether *R* is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in *F* is a superkey (by computing the attribute closure and seeing if it includes all attributes of *R*).
- Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: \Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key." If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure 15.7, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)

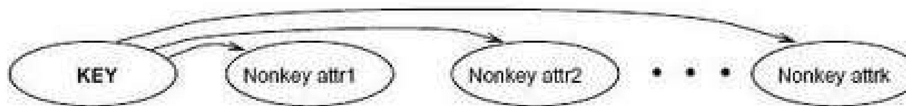


Figure FDs in a BCNF Relation

- BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 15.8.
  X Y A
  X y_1 *a*
  x y_2 ?
- **Figure 15.8** Instance Illustrating BCNF
  This figure shows (two tuples in) an instance of a relation with three attributes *X*, *Y*, and *A*. There are two tuples with the same value in the *X* column. Now suppose that we know that this instance satisfies an FD *X ! A*. We can see that one of the tuples has the value *a* in the *A* column. What can we infer about the value in the *A* column in the second tuple? Using the FD, we can conclude that the second tuple also has the value *a* in this column. (Note that this is really the only kind of inference we can make about values in the fields of tuples by using FDs.) But isn't this situation an example of redundancy? We appear to have stored the value *a* twice. Can such a situation arise in a BCNF relation? No! If this relation is in BCNF, because *A* is distinct from *X* it follows that *X* must be a key. (Otherwise, the FD *X ! A* would violate BCNF.) If *X* is a key, then $y_1 = y_2$, which means that the two tuples are identical. Since a relation is defined to be a *set* of tuples, we cannot have two copies of the same tuple and the situation shown in Figure 15.8 cannot arise. Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot

be inferred (using only FDs) from the values in all other fields in (all tuples of) the relation instance.

Q. write a short note on Third Normal Form?

- **Third Normal Form**
- Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and *A* be an attribute of *R*. *R* is in **third normal form** if for every FD *X ! A* that holds over *R*, one of the following statements is true: *A 2 X*; that is, it is a trivial FD, or *X* is a superkey, or *A* is part of some key for *R*.
- The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a *minimal* set of attributes that uniquely determines all other attributes. *A* must be part of a key (any key, if there are several).
- It is not enough for *A* to be part of a superkey, because the latter condition is satisfied by each and every attribute! Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.
- Suppose that a dependency *X ! A* causes a violation of 3NF. There are two cases: X *is a proper subset of some key* K. Such a dependency is sometimes called a **partial dependency**. In this case we store *(X, A)* pairs redundantly. As an example, consider the Reserves relation with attributes *SBDC* from Section 15.3.4.
- The only key is *SBD*, and we have the FD *S ! C*. We store the credit card number for a sailor as many times as there are reservations for that sailor. X *is not a proper subset of any key.* Such a dependency is sometimes called a **transitive dependency** because it means we have a chain of dependencies *K ! X ! A*. The problem is that we cannot associate an *X* value with a *K* value unless we also associate an *A* value with an *X* value. As an example, consider the Hourly Emps relation with attributes *SNLRWH* from Section 15.3.1. The only key is *S*, but there is an FD *R ! W*, which gives rise to the chain *S ! R ! W*. The consequence is that we cannot record the fact that employee *S* has rating *R* without knowing the hourly wage for that rating. This condition leads to insertion, deletion, and update anomalies.
- Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are illustrated in Figure 15.10. Note that in Figure 15.10, the set *X* of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that *X* is not a subset of KEY.

Case 1: A not in KEY

Fig. Partial Dependencies

- The motivation for 3NF is rather technical. By making an exception for certain de- pendencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties (Section 15.6). Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible. We may therefore compromise by settling for a 3NF design.
- As we shall see in Chapter 16, we may sometimes accept this compromise (or even settle for a non-3NF schema) for other reasons as well.
- Unlike BCNF, however, some redundancy is possible with 3NF. The problems associ- ated with partial and transitive dependencies persist if there is a nontrivial dependency *X ! A* and *X* is not a superkey, even if the relation is in 3NF because *A* is part of a key. To understand this point, let us revisit the Reserves relation with attributes *SBDC* and the FD *S ! C*, which states that a sailor uses a unique credit card to pay for reservations. *S* is not a key, and *C* is not part of a key. (In fact, the only key is *SBD*.) Thus, this relation is not in 3NF; *(S, C)* pairs are stored redundantly. However, if we also know that credit cards uniquely identify the owner, we have the FD *C ! S*, which means that *CBD* is also a key for Reserves. Therefore, the dependency *S ! C* does not violate 3NF, and Reserves is in 3NF. Nonetheless, in all tuples containing the same *S* value, the same *(S, C)* pair is redundantly recorded. For completeness, we remark that the definition of

second normal form is essentially that partial dependencies are not allowed. Thus, if a relation is in 3NF (which precludes both partial and transitive dependencies), it is also in 2NF.

Q. Define lossless-Join decomposition with example?

- Let R be a relation schema and let F be a set of FDs over R. A decomposition of R into two schemas with attribute sets X and Y is said to be a lossless-join decomposition with respect to F if, for every instance r of R that satisfies the dependencies in F, πx(r) ⋈ πy(r) = r. In other words, we can recover the original relation from the decomposed relations.
- This definition can easily be extended to cover a decomposition of R into more than two relations. It is easy to see that r ⊆ πx(r) ⋈ πy(r) always holds. In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 19.9.
- By replacing the instance r shown in Figure 19.9 with the instances πSP(r) and πPD(r), we lose some information. In particular, suppose that the tuples in r denote relationships. We can no longer tell that the relationships d3) and (s3, p1, d1) do not hold. The decomposition of schema SPD into SP and PD is therefore lossy if the instance r shown in the figure is legal, that is, if this

Instance r

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3

$\pi_{SP}(r)$

S	P
s1	p1
s2	p2
s3	p1

$\pi_{PD}(r)$

P	D
p1	d1
p2	d2
p1	d3

$\pi_{SP}(r) \bowtie \pi_{PD}(r)$

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3
s1	p1	d3
s3	p1	d1

Figure 19.9 Instances Illustrating Lossy Decompositions

- instance could arise in the enterprise being modeled

All decompositions used to eliminate redundancy must be lossless. The following simple test is very useful:

In other words, the attributes common to *R*1 and *R*2 must contain a key for either *R*1 or *R*2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join.2 If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set) algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it. Consider the Hourly Emps relation again. It has attributes *SNLRWH*, and the FD

R ! W causes a violation of 3NF. We dealt with this violation by decomposing the relation into *SNLRH* and *RW*. Since *R* is common to both decomposed relations, and *R ! W* holds, this decomposition is lossless-join.
This example illustrates a general observation: If an FD *X!Y* holds over a relation *R* and *X\Y* is empty, the decomposition of *R* into *R − Y* and *XY* is lossless. *X* appears in both *R−Y* (since *X \ Y* is empty) and *XY*, and it is a key for *XY*. Thus, the above observation follows from the test for a lossless-join decomposition.
Another important observation has to do with repeated decompositions. Suppose that a relation *R* is decomposed into *R1* and *R2* through a lossless-join decomposition, and that *R1* is decomposed into *R11* and *R12* through another lossless-join decomposition.
Then the decomposition of *R* into *R11*, *R12*, and *R2* is lossless-join; by joining *R11* and *R12* we can recover *R1*, and by then joining *R1* and *R2*, we can recover *R*.

Quick Revision

- *Redundancy*, storing the same information several times in a database, can result in *update anomalies* (all copies need to be updated), *insertion anomalies* (certain information cannot be stored unless other information is stored as well), and *deletion anomalies* (deleting some information means loss of other information as well). We can reduce redundancy by replacing a relation schema *R* with several smaller relation schemas. This process is called *decomposition*.
- A *functional dependency X ! Y* is a type of IC. It says that if two tuples agree upon (i.e., have the same) values in the *X* attributes, then they also agree upon the values in the *Y* attributes. FDs can help to refine subjective decisions made during conceptual design.
- An FD *f is implied by* a set *F* of FDs if for all relation instances where *F* holds,*f* also holds. The closure of a set *F* of FDs is the set of all FDs F^+ implied by *F*. *Armstrong's Axioms* are a sound and complete set of rules to generate all FDs in the closure. An FD *X ! Y* is *trivial* if *X* contains only attributes that also appear in *Y*. The *attribute closure* X^+ of a set of attributes *X* with respect to a set of FDs *F* is the set of attributes *A* such that *X ! A* can be inferred using Armstrong's Axioms.
- A *normal form* is a property of a relation schema indicating the type of redundancy that the relation schema exhibits. If a relation schema is in *Boyce-Codd normal form* (BCNF), then the only nontrivial FDs are key constraints. If a relation is in third normal form (3NF), then all nontrivial FDs are key constraints or their right side is part of a candidate key. Thus, every relation that is in BCNF is also in 3NF, but not vice versa.
- A decomposition of a relation schema *R* into two relation schemas *X* and *Y* is a *lossless-join* decomposition with respect to a set of FDs *F* if for any instance *r* of *R* that satisfies the FDs in *F*, $\pi_X(r) \bowtie \pi_Y(r) = r$. The decomposition of *R* into *X* and *Y* is lossless-join if and only if F^+ contains either *X* *Y ! X* or the
- FD *X \ Y ! Y* . The decomposition is *dependency-preserving* if we can enforce all FDs that are given to hold on *R* by simply enforcing FDs on *X* and FDs on *Y* independently (i.e., without joining *X* and *Y*).

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain functional dependencies?	27
2	Q. write a short note on Boyce-Codd Normal Form?	28
3	Q. write a short note on Third Normal Form?	29
4	Q. Define lossless-Join decomposition with example?	30