D.N.R.COLLEGE (AUTONOMOUS),BHIMAVARAM

M.Sc(CS) DEPARTMENT

RALATIONAL DATABASE MANAGEMENT SYSTEMS

I M.Sc(CS)

Presented by

V.SARALA

M.Sc.(CS) DEPARTMENT

RELATIONAL DATABASE MANAGEMENT SYSTEMS (MSCS 204)

Theory : 4 Periods **Ext. Marks : 75**

Lab Hrs : 3 Periods **Mid Marks : 25**

Exam : 3 Hrs. **Credits : 4**

UNIT I

Database System Introduction: DBMS, Database Users, Data Models, Schemas and Instances, Three-Schema Architecture and Data Independence, The Database System Environment, Centralized and Client/Server Architectures

Data Modeling Using the Entity-Relationship Model: Using High-Level Conceptual Data Models for Database Design, Entity Types, Entity Sets Attributes and Keys, Relationships Types, Relationship Sets, Weak Entity types, ER diagrams

Relational Data Model and Relational Database Constraints: Relational Model Concepts, Constraints and Relational Schemas, Update Operations and Dealing with Constraint Violations, Relational Database Design Using ER to Relational Mapping.

UNIT II

Relational Database Design: Design Guidelines for Relation Schema, Functional Dependencies, Normal Forms Based on Primary keys, Definitions of Second and Third Normal forms, BCNF, Properties of Relational Decomposition, Algorithm for Relational Database Design

Indexing Structures for files: types of single level ordered indexes, multilevel indexes, dynamic multilevel indexes using B Trees and B + Trees, Indexes on multiple keys.

UNIT III

Transaction Processing: Transaction and System Concepts, Desirable Characteristics of Transactions, Characterising Schedules Based On Recoverability and Serializability

Concurrency Control Techniques: Two Phase Locking, Timestamp Ordering, Validation Concurrency Control, Multiple Granularity Locking

Database Recovery Techniques: Recovery Concepts, Recovery Based On Deferred and Immediate Updates, Shadow Paging

UNIT IV

Object Relational Systems: Object Relational Support in SQL, Encapsulation of Operations in SQL, Inheritance and Overloading of Functions in SQL.

Distributed Databases: Advantages of Distributed Databases, Data FragmentationReplication and Allocation Techniques, Types of Distributed Systems, Query Processing in Distributed Databases, Concurrency Control and Recovery in Distributed databases.

TEXT BOOK:

1. Fundamentals of Database System,Elmasri, Navathe, Pearson Educaiton.

REFERENCES BOOKS:

1. Database Management Systems, Raghu Ramakrishnan,JohannesGehrke,McGraw-Hill.
2. Database Concepts, Abraham Silberschatz, Henry F Korth, S Sudarshan, McGraw-Hill

RELATIONAL DATABASE MANAGEMENT SYSTEMS

UNIT - I

Database System Introduction

DBMS

Databases and database technology having a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, social media, engineering, medicine, genetics, law, education, and library science.

Definition: A '**database'** is a collection of related data. ('data' as both singular and plural in database literature. In standard English, 'data' is used only for plural, 'datum' is used for singular).

By '**data'**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. Nowadays, this data is typically stored in mobile phones, which have their own simple database software. This data can also be recorded in an indexed address book or stored on a hard drive, using a personal computer and software such as Microsoft Access or Excel. This collection of related data with an implicit meaning is a database.

The common use of the term *database* is usually more restricted. A database has the following implicit properties:

1. A database represents some aspect of the real world, sometimes called the **miniworld** or the **universe of discourse (UoD)**. Changes to the miniworld are reflected in the database.
2. A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
3. A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

Definition: A **database management system (DBMS)** is a computerized system that enables users to create and maintain a database. The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications.

Defining: a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called **meta-data**.

Constructing: the database is the process of storing the data on some storage medium that is controlled by the DBMS.

Manipulating: a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

Sharing: a database allows multiple users and programs to access the database simultaneously.

Other important functions provided by the DBMS include protecting the database and maintaining it over a long period of time.

Protection: includes system protection against hardware or software malfunction (or crashes) and security protection against unauthorized or malicious access.

Maintain: A typical large database may have a life cycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.

To complete our initial definitions, we will call the database and DBMS software together a **database system**.

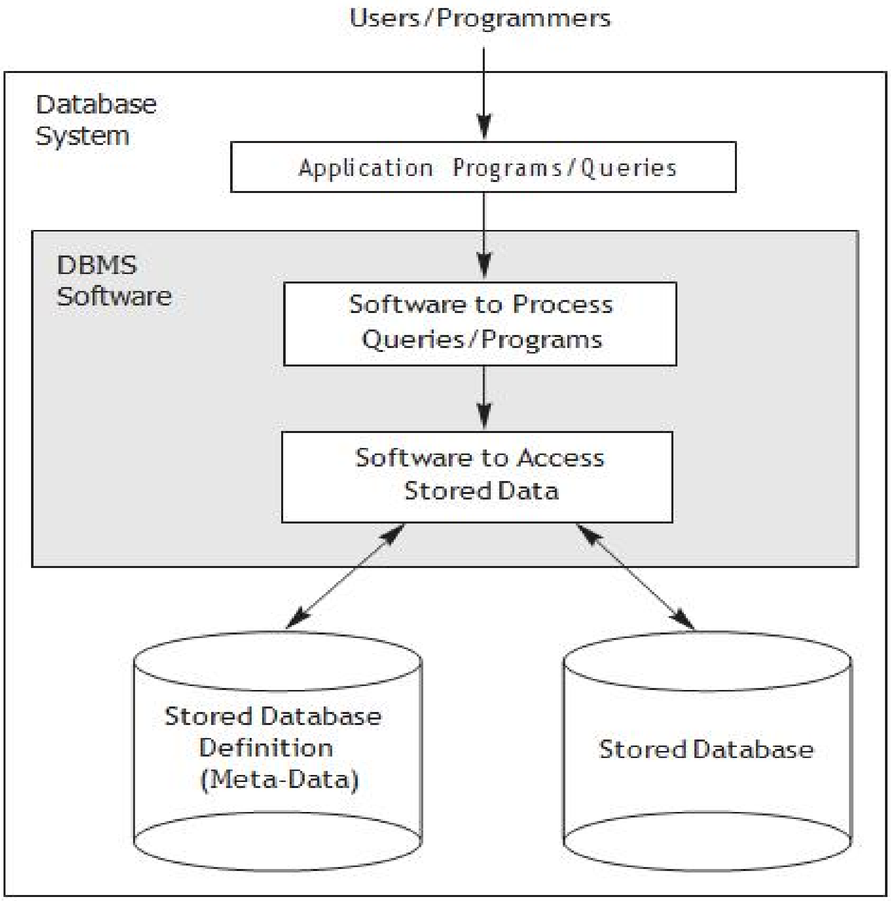
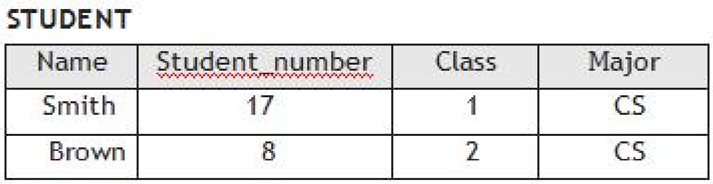

Users/Programmers
Database System
Application Programs/Queries
DBMS Software
Software to Process Queries/Programs
Software to Access Stored Data
Stored Database Definition (Meta-Data)
Stored Database

Figure 1.1
A simplified database system environment.

An Example

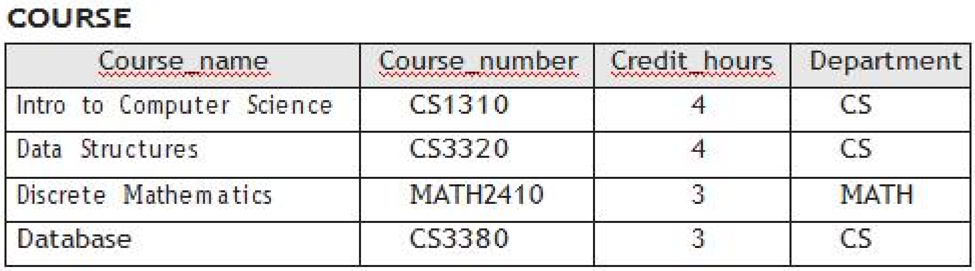
Let us consider a simple example that most readers may be familiar with: a **UNIVERSITY** database for maintaining information concerning students, courses, and grades in a university environment. The database is organized as five files, each of which stores **data records** of the same type.
The STUDENT file stores data on each student,
the COURSE file stores data on each course,
the SECTION file stores data on each section of a course,
the GRADE_REPORT file stores the grades that students
receive in the various sections they have completed, and
the PREREQUISITE file stores the prerequisites of each course.

To define this database, we must specify the structure of the records of each file by specifying the different types of ‘data elements’ to be stored in each record.



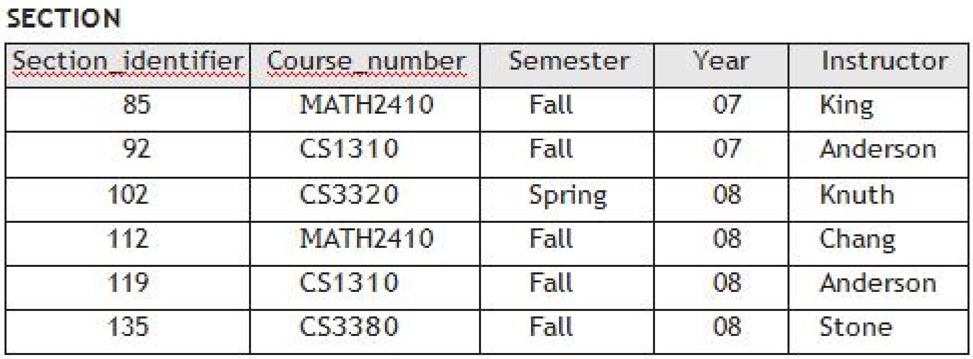
STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS



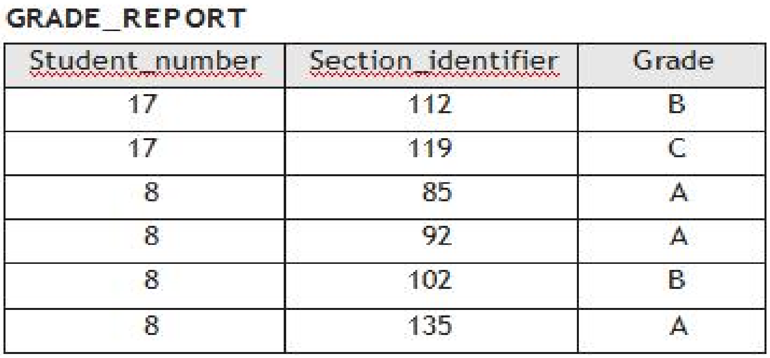
COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS



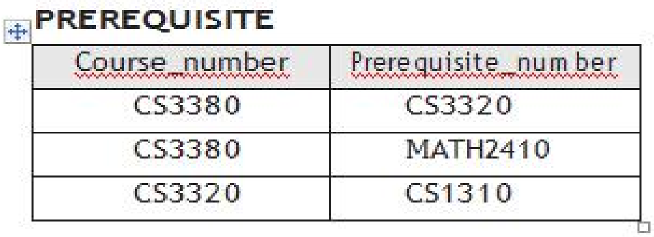
SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone



GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A



PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Characteristics of the Database Approach

File processing: each user defines and implements the files needed for aspecific software application as part of programming the application.

The main characteristics of the database approach versus the file processing approach are the following:

1. Self-describing nature of a database system
2. Insulation between programs and data, and data abstraction
3. Support of multiple views of the data
4. Sharing of data and multiuser transaction processing

1. Self-Describing Nature of a Database System:

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description ofthe database structure and constraints. This definition is stored in the DBMS cata- log, which contains information such as the structure of each file, the type and stor- age format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data**, and it describes the structure of the pri- mary database.

2. Insulation between Programs and Data,and Data Abstraction:

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require '*changing all programs*' that access that file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property '**program-data independence**'.

For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure:

Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
Student_number	31	4
Class	35	1
Major	36	4

Fig: Internal Storage format for a Student record.

If we want to add another piece of data to each STUDENT record, say the Birth_date, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we only need to change the *description* of STUDENT records in the catalog (Figure) to reflect the inclusion of the new data item Birth_date; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

An **operation** (also called a *function* or *method*) is specified in two parts:

The ***interface*** **(or *signature*)** of an operation includes the operation name and the data types of its arguments (or parameters).

The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface.

User application programs can operate on the data by invoking these operations through their names and arguments are called '**program-operation independence'**.

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored or how the operations are implemented.

A **data model** is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships,

3. Support of Multiple Views of the Data:

A database typically has many types of users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether, the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views.

4 Sharing of Data and Multiuser Transaction Processing:

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct.

For example, when several reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger. These types of applications are generally called **online transaction processing (OLTP)**.

The concept of a **transaction** has become central to many database applications. A transaction is an executing program or process that includes one or more database accesses, such as reading or updating of database records. The DBMS must enforce several transaction properties.

Actors on the Scene

In large organizations, many people are involved in the design, use, and maintenance of a large database with hundreds or thousands of users. The people whose jobs involve the day-to-day use of a large database; we call them the actors on the scene.

1. Database Administrators : In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA).** The DBA is responsible for authorizing access to the database, coordinating and monitoring its use.

2. Database Designers: Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data.

3. End Users: End users are the people whose jobs require access to the database for querying, updating, and generating reports.

There are several categories:

1. Casual end users: occasionally access the database, but they may need different information each time.
2. Naive or parametric end users: make up a sizable portion of database end users.
3. Sophisticated end users: include engineers, scientists, business analysts, and others are the facilities of the DBMS as to implement their own applications.
4. Stand alone users: maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces.

4. System Analysts and Application Programmers (Software Engineers): System analysts determine the requirements of end users. Application programmers implement these specifications as programs; then they test, debug, document, and maintain these canned transactions.

Advantages of Using the DBMS Approach

1. Controlling Redundancy: The redundancy in storing the same data multiple times leads to several problems. The DBMS should have the capability to control this redundancy. So as to prohibit, inconsistencies among the files.

2. Restricting Unauthorized Access: When multiple users share a large database, that most users will not be authorized to access all information in the database. A DBMS should provide a **security** and **authorization subsystem**, which the DBA uses to create accounts and to specify account restrictions.

3. Providing Persistent Storage for Program Objects: This is one of the main reasons for object-oriented database systems. Programming languages typically have complex data structures, such as record types in Pascal or class definitions in C++ or Java.

4. Providing Storage Structures for Efficient Query Processing: Database systems must provide capabilities for efficiently executing queries and updates. The DBMS must provide specialized data structures to speed up disk search for the desired records.

5. Providing Backup and Recovery: A DBMS must provide facilities for recovering from hardware or software failures. The backup and recovery subsystem of the DBMS is responsible for recovery.

6. Providing Multiple User Interfaces: Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include apps for mobile users, query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for standalone users. Both forms-style interfaces and menu-driven interfaces are commonly known as graphical user interfaces (GUIs).

7. Representing Complex Relationships among Data: A DBMS must have the capability to represent a variety of complex relationships among the database as well as to retrieve and update related data easily and efficiently.

8. Enforcing Integrity Constraints: Most database applications have certain integrity constraints that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints.

9. Permitting Inferencing and Actions Using Rules: Some database systems provide capabilities for defining deduction rules for inferencing new information from the stored database facts. Such systems are called **deductive database systems**.

10. Additional Implications of Using the Database Approach:

1. Potential for Enforcing Standards.
2. Reduced Application Development Time.
3. Flexibility
4. Availability of Up-to-Date Information.
5. Economies of Scale

A Brief History of Database Applications

1. Early Database Applications Using Hierarchical and Network Systems
2. Providing Data Abstraction and Application Flexibility with Relational Databases
3. Object-Oriented Applications and the Need for More Complex Databases
4. Interchanging Data on the Web for E-Commerce
5. Extending Database Capabilities for New Applications

Database System Concepts and Architecture

The architecture of DBMS package has evolved from the early monolithic systems, where the whole DBMS software package was one lightly integrated system, to the modern DBMS packages are modular in design, with a client/server system architecture.

Data Models, Schemas, and Instances

Data Model: A data model is a collection of concepts that can be used to describe the structure of a database. By structure of a database we mean the data types, relationships, and constraints that apply to the data. Most data models also include a set of 'basic operations' for specifying retrievals and updates on the database.

Categories of Data Models

We can categorize according to the types of concepts they use to describe the database structure. 'High-level' or 'conceptual data models' provide concepts that are close to the way many users perceive data, whereas 'low-level' or 'physical data models' provide concepts that describe the details of how data is stored on the computer.

Between these two extremes is a class of 'representational (or implementation) data models', which provide concepts that may be easily understood by end users.

Conceptual data models use concepts such as entities, attributes, and relationships. An **'entity**' represents a real-world object or concept, such as an employee or a project, that is described in the database. An '**attribute**' represents some property of interest that further describes an entity, such as the employee's name or salary. A '**relationship**' among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project.

Representational data models represent data by using record structures and hence are sometimes called record-based data models.

'**Object data model'** as a new family of higher-level implementation data models that are closer to conceptual data models.

Schemas, Instances, and Database State

In a data model, it is important to distinguish between the 'description' of the database and the 'database itself'. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently.

Most data models have certain conventions for displaying schemas as diagrams. A displayed schema is called a **schema diagram.**

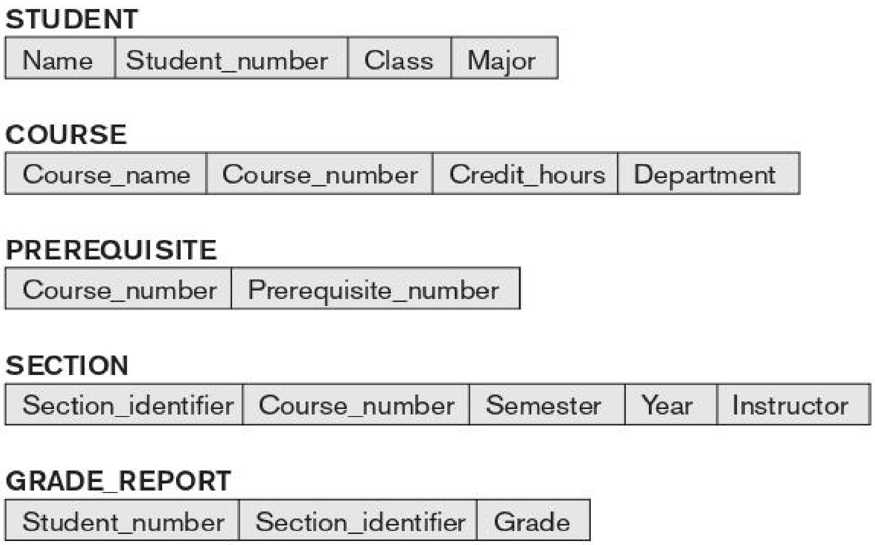


Figure 2.1
Schema diagram for the database in Figure 1.2.

STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

PREREQUISITE

Course_number	Prerequisite_number

SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

A schema diagram displays only some aspects of a schema, such as the names of record types and data items. Other aspects are not specified in the schema diagram.

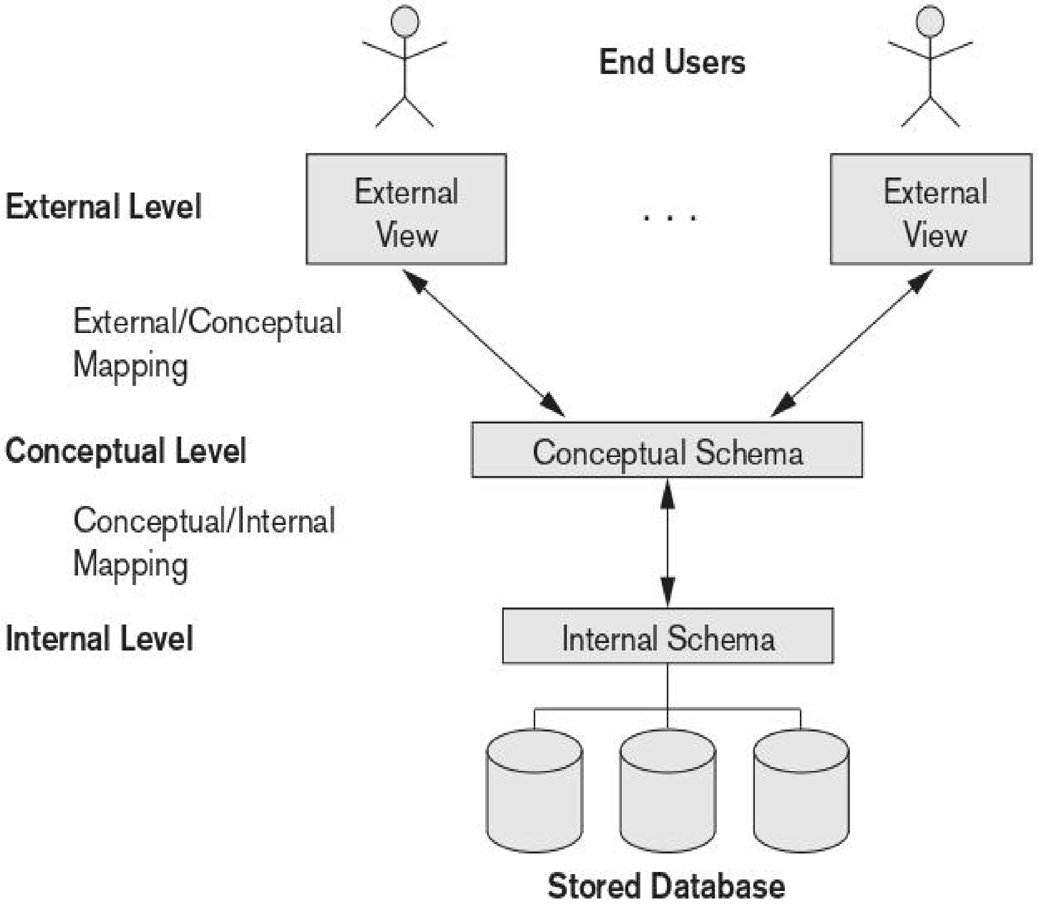
The data in the database at a particular moment in time is called a **database state** or **snapshot**. It is also called the current set of **occurrences** or **instances** in the database. In a given database state, each schema construct has its own current set of instances. For example, the STUDENT construct will contain the set of individual student entities (records) as its instances.

The DBMS stores the descriptions of the schema constructs and constraints also called the **meta-data.**

Three-Schema Architecture and Data Independence

The Three-Schema Architecture:

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications from the physical database.



Schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints.

3. The **external** or **view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group.

The processes of transforming requests and results between levels are called **mapping.**

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. Changes to constraints can be applied 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 schema. Hence, the external schemas need not be changed as well. For example, by creating additional access structures to improve the performance of retrieval or update.

The Database System Environment

1. DBMS Component Modules

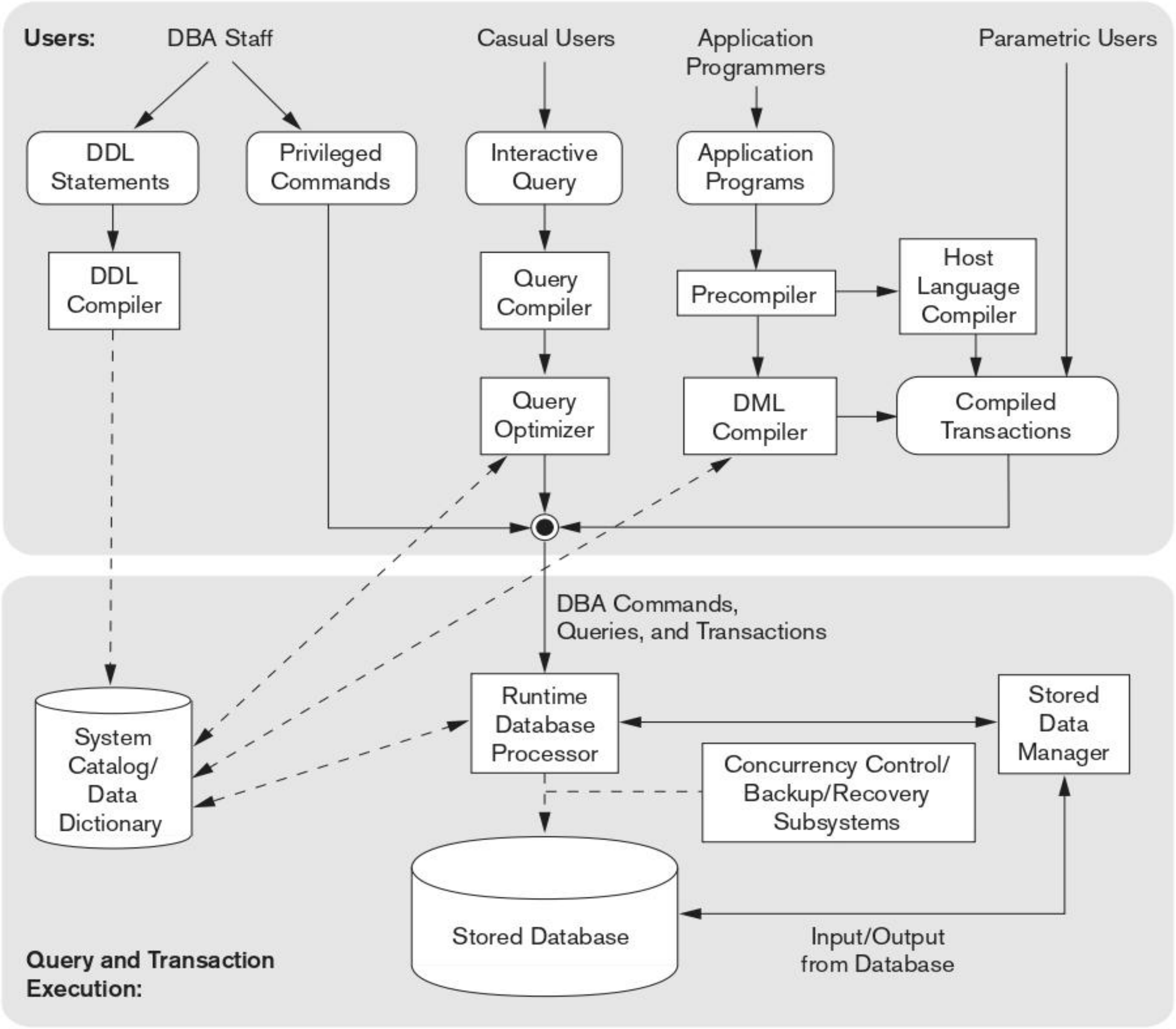


Figure 2.3
Component modules of a DBMS and their interactions.

The Figure illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internal modules of the DBMS responsible for storage of data and processing of transactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the operating system (OS), which schedules disk read/write. A higher-level **‘stored data manager’** module of the DBMS controls access to DBMS information that is stored on disk.

Let us consider the top part of Figure first. It shows interfaces for the DBA staff, **casual users** who work with interactive interfaces to formulate queries, **application programmers** who create programs using some host programming languages, and **parametric users** who do data entry work by supplying parameters to predefined transactions.

The DBA staff works on defining the database using the DDL and other privileged commands. The **DDL compiler** processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog.

Casual users and persons with occasional need for information from the database interact using the **interactive query** interface. These queries are parsed and validated for correctness of the query syntax, the names of files and data elements, and so on by a **query compiler** that compiles them into an internal form. This internal query is subjected to query optimization. Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the **DML compiler** for compilation into object code for database access.

In the lower part of Figure, the **runtime database processor** handles database accesses at run time, it receives retrieves or update operations and carries them out on the database. It works with the **system catalog** and may update it with statistics. It also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory.

We have shown **concurrency control and backup and recovery systems** separately as a module in this figure. They are integrated into the working of the runtime database processor for purposes of transaction management.

The **client program** accesses the DBMS running on a separate computer or device from the computer on which the database resides. The former is called the **client computer,** and the latter is called the **database server**. In many cases, the client accesses a middle computer, called the **application server**, which in turn accesses the database server.

2. Database System Utilities

Most DBMSs have **database utilities** that help the DBA manage the database system. Common utilities have the following types of functions:

Loading: A loading utility is used to load existing data files such as text files or sequential files into the database.

Backup: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. Incremental backups are also often used.

File reorganization: This utility can be used to reorganize a set of database files into different file organization to improve performance.

Performance monitoring: Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.

3. Tools, Application Environments, and Communications Facilities:

Other tools are often available to database designers, users, and the DBMS. CASE tools are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or data repository) **system**. The data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository**.

Application development environments, such as PowerBuilder (Sybase) or JBuilder (Borland), have been quite popular. These systems provide an environment for developing database applications.

The DBMS also needs to interface with **communications software**, whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or personal computers.

The integrated DBMS and data communications system is called a DB/DC system. Communications networks are needed to connect the machines. These are often local area networks (LANs).

Centralized and Client/Server Architectures for DBMSs

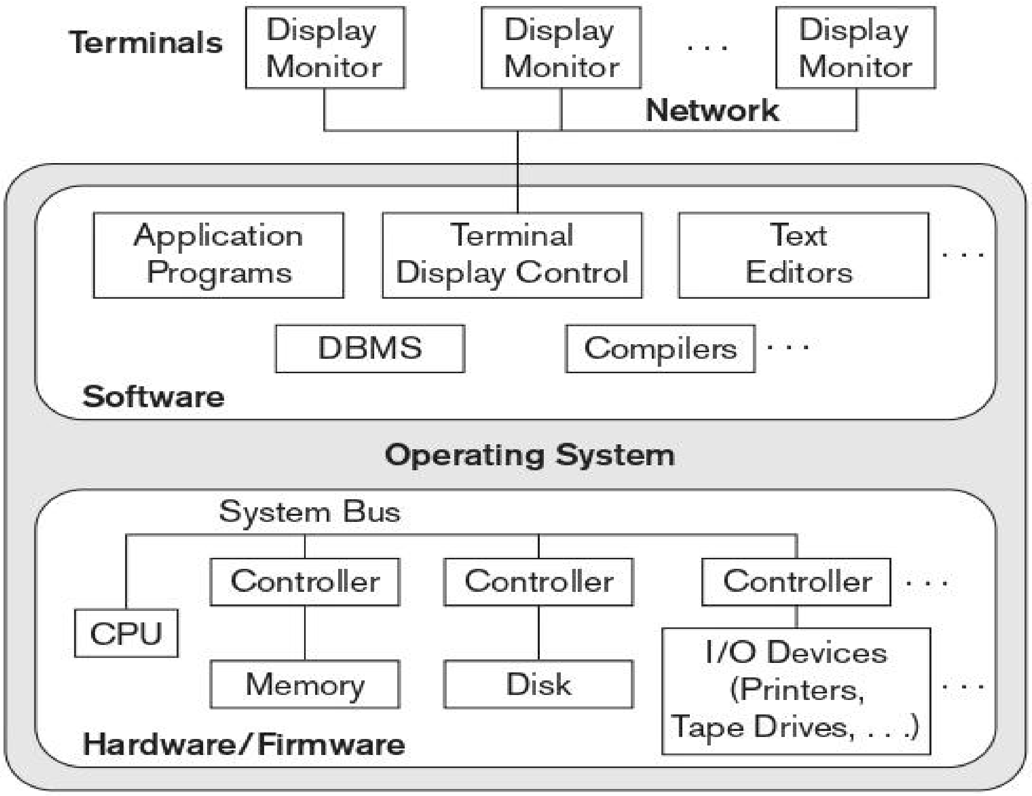
1. Centralized DBMSs Architecture:

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Older architectures used mainframe computers to provide the main processing for all system.

All processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers in the same way as they had used display terminals, so that the DBMS itself was still a **centralized DBMS** in which all the DBMS functionality application program execution, and user interface processing were carried out on one machine.

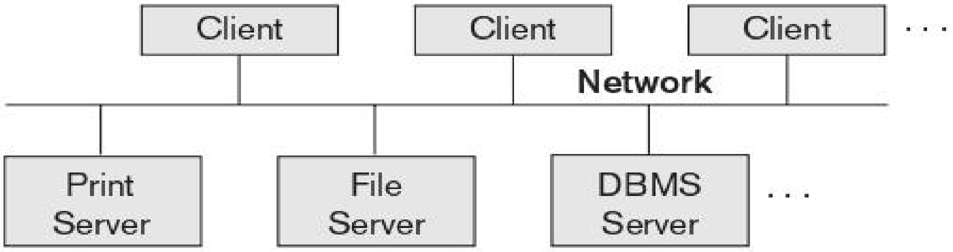
Figure: A Physical Centralized architecture



2. Basic Client/Server Architectures

The client/server architecture was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network.

Figure: Logical two tier client/server architecture



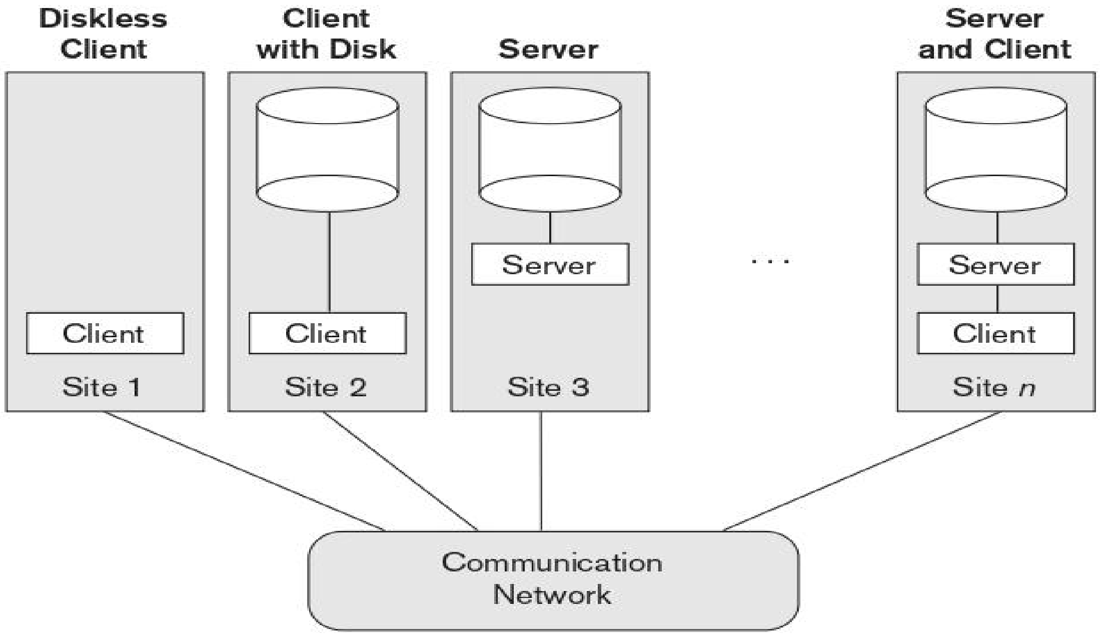
A **file server** that maintains the files of the client machines. A **printer server** connected to various printers; thereafter, all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** are fall into the specialized server category. **Specialized servers** can be accessed by many client machines.

The **client machines** provide the user with the appropriate interfaces to utilize these servers.

A **client** in this framework is typically a user machine that provides user interface capabilities and local processing.

3. Two-Tier Client/Server Architectures for DBMSs

Figure: Physical Two- Tier Client/Server Architecture for DBMSs



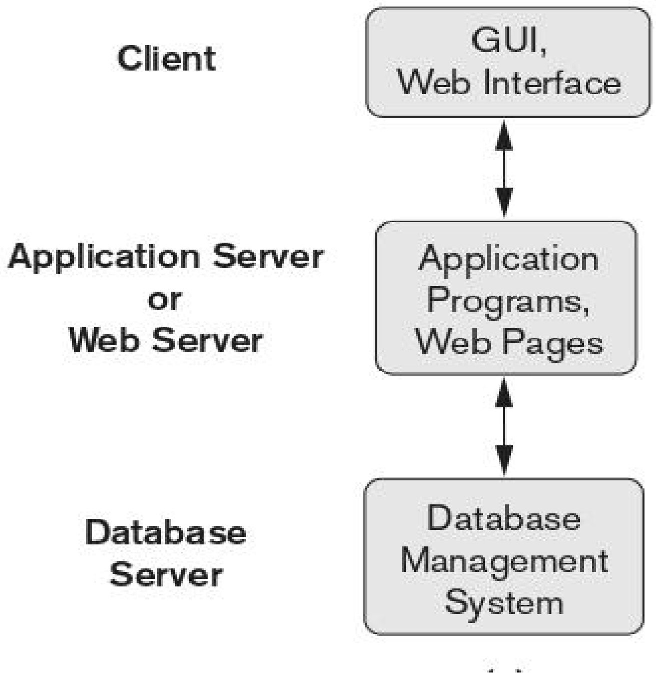
The Client/Server architecture is increasingly being incorporated into commercial DBMS packages. In relational database management systems (RDBMSs), many of which started as centralized systems. Because SQL provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server.**

When DBMS access is required, the program establishes a connection to the DBMS, once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity** (**ODBC**) provides an **application programming interface** (**API**), which allows client-side programs.

The architectures described here are called **two-tier architectures** because the software components are distributed over two systems: **client** and **server**. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

4. Three-Tier and n-Tier Architectures for Web Applications

Figure: Logical Three-tier Client/Server architecture


Client
GUI, Web Interface
Application Server or Web Server
Application Programs, Web Pages
Database Server
Database Management System

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure.

This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by running application programs and storing business rules that are used to access data from the database server.

Clients contain GUI interfaces and Web browsers. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to the users. Thus, the **user interface**, **application rules,** and **data access** act as the three tiers.

Data Modeling Using the Entity– Relationship (ER) Model

The modeling concepts of the entity–relationship (ER) model, which is a popular high-level conceptual data model.

Object modeling methodologies such as the Unified Modeling Language (UML) are becoming increasingly popular in both database and software design.

Using High-Level Conceptual Data Models for Database Design

The main phases of database design:

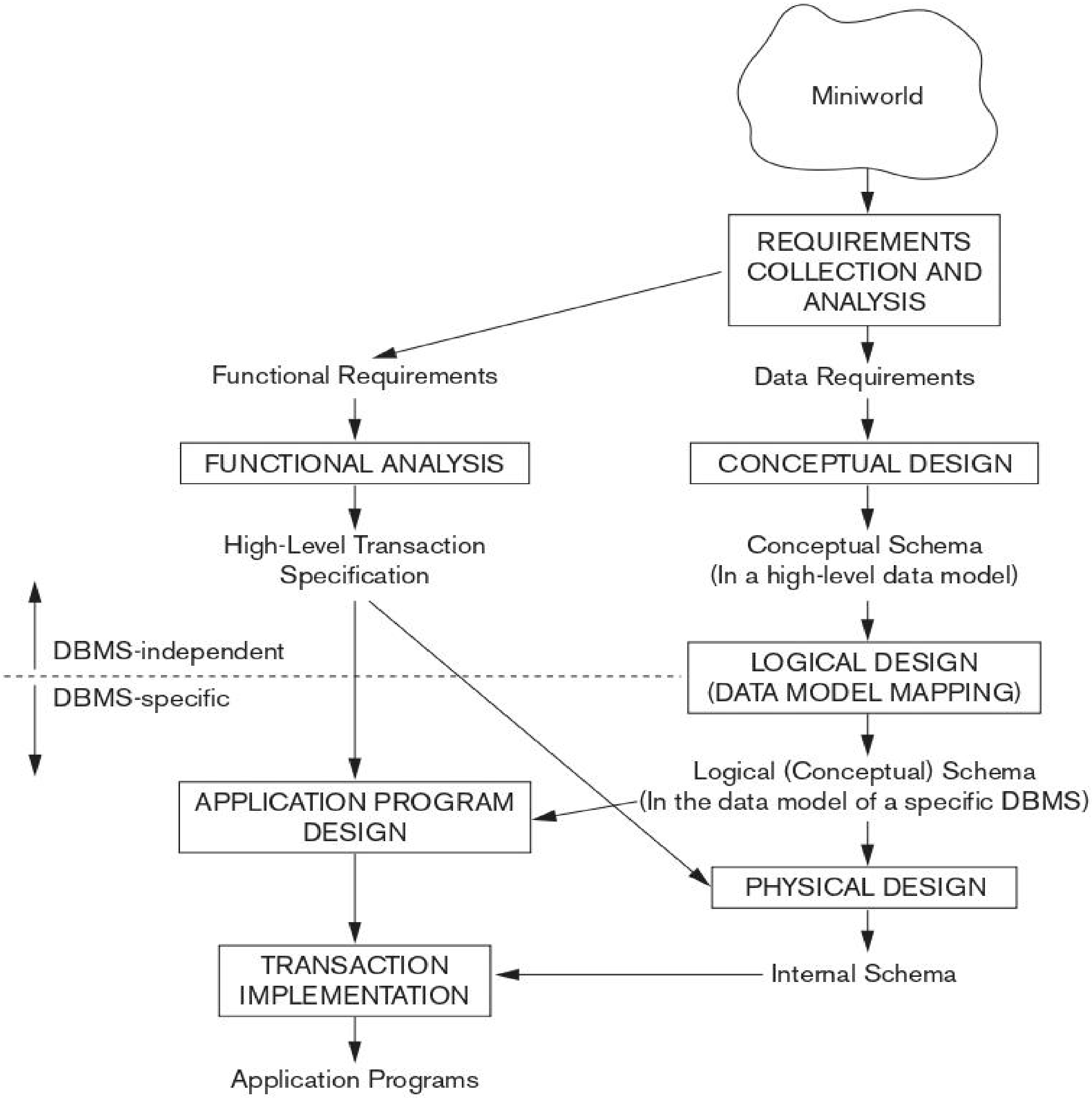

Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Data Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

Figure shows a simplified description of the database design process. The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements.

In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user defined operations (or transactions) that will be applied to the database, including both retrievals and updates.

Once all the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called conceptual design. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational (SQL) model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**, and its result is a database schema in the implementation data model of the DBMS.

The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths for the database files are specified. In parallel, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications.

Entity Types, Entity Sets, Attributes, and Keys

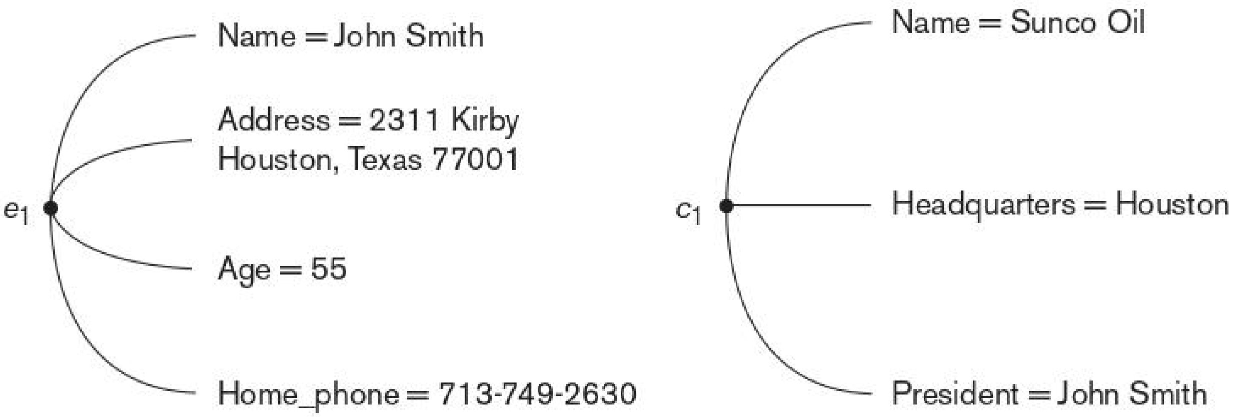
The ER model describes data as entities, relationships, and attributes.

1. **Entities and Attributes:**

Entities and Their Attributes. The basic concept that the ER model represents is an **entity**, which is a **thing** or **object** in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).

Each entity has attributes—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.

Figure: Two entities, e1 and company c1, and their attributes



The EMPLOYEE entity e1 has four attributes: Name, Address, Age, and Home_phone; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001', '55', and '713-749-2630', respectively.

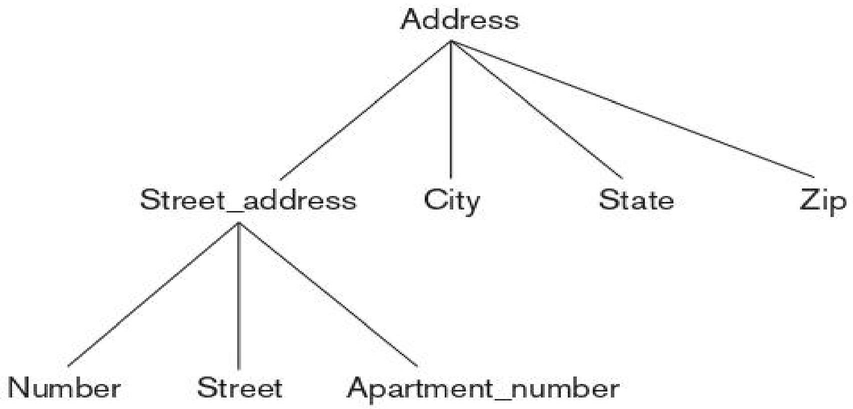
The COMPANY entity c1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively.

Several types of attributes occur in the ER model: simple versus composite, single- valued versus multivalued, and stored versus derived.

Composite versus Simple (Atomic) Attributes: Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure can be subdivided into Street_address, City, State, and Zip, with the values '2311 Kirby', 'Houston', 'Texas', and '77001'.

Attributes that are not divisible are called **simple or atomic attributes**.

Fig: A hierarchy of composite attributes



Single-Valued versus Multivalued Attributes: Most attributes have a single value for a particular entity, such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity. For eg, a College_degrees attribute for a person, one person may not have any college degrees, another person may have one, and a third person may have two or more degrees. Such attributes are called **multivalued.**

Stored versus Derived Attributes: In some cases, two (or more) attribute values are related, for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a **derived attribute** and is said to be derivable from the Birth_date attribute, which is called a **stored attribute**.

NULL Values: In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. For such situations, a special value called **NULL** is created.

Complex Attributes: Notice that, in general, composite and multivalued attributes can be nested arbitrarily. We can represent arbitrary nesting by grouping components of a composite attribute between parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces { }. Such attributes are called **complex attributes**.

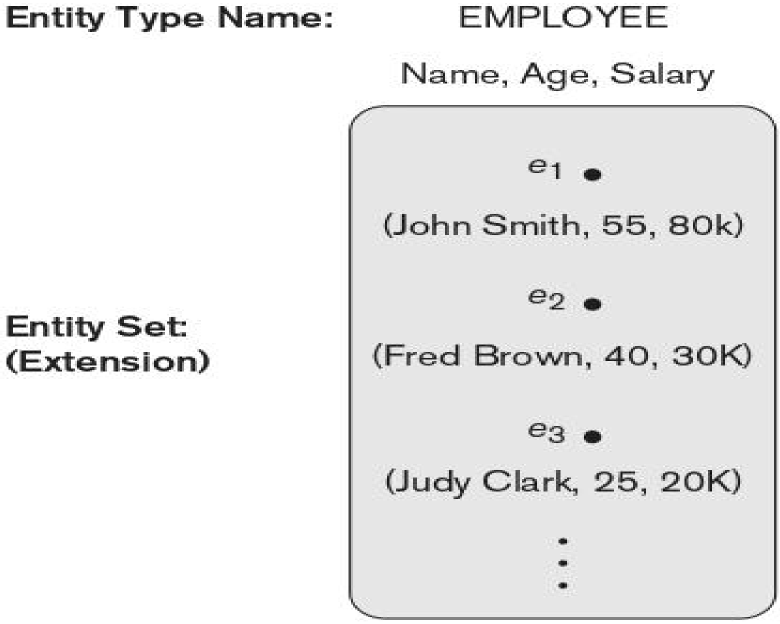
Eg: {Address_phone({Phone(Area_code,Phone_number)},Address(Street_address (Number,Street,Apartment_number),City,State,Zip))}

2. Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets: A database usually contains groups of entities that are similar An **entity type** defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. Figure shows two entity types: EMPLOYEE and COMPANY, and a list of attributes for each.

The collection of all entities of a particular entity type in the database at any point in time is called an **entity set**; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a type of entity as well as the current 'set of all employee entities' in the database.

Eg:



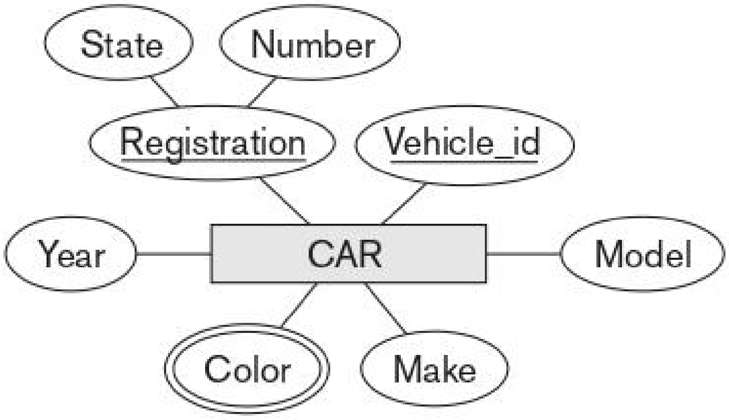
:

An entity type is represented in ER diagrams as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

Key Attributes of an Entity Type: An important constraint on the entities of an entity type is the key or uniqueness constraint on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a key attribute, and its values can be used to identify each entity uniquely.

Value Sets (Domains) of Attributes: A value set, which specifies the set of values that may be assigned to that attribute for each individual entity. In the above Figure, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute to be the set of strings of alphabetic characters separated by blank characters, and so on. Value sets are typically specified using the basic 'data types'.

Eg: ER diagram for the CAR entity type with two key attributes, Registration and Vehicle_id.

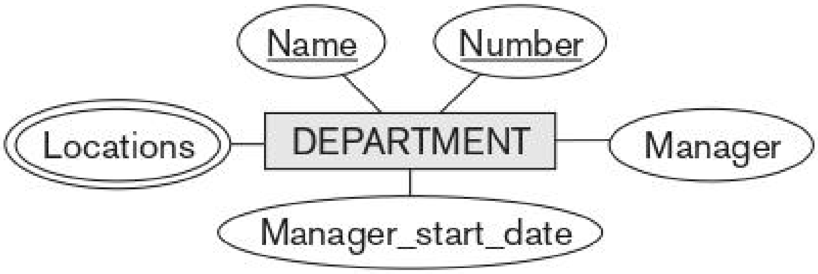


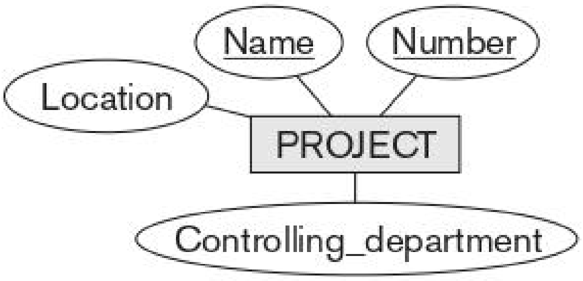
3. Initial Conceptual Design of the COMPANY Database

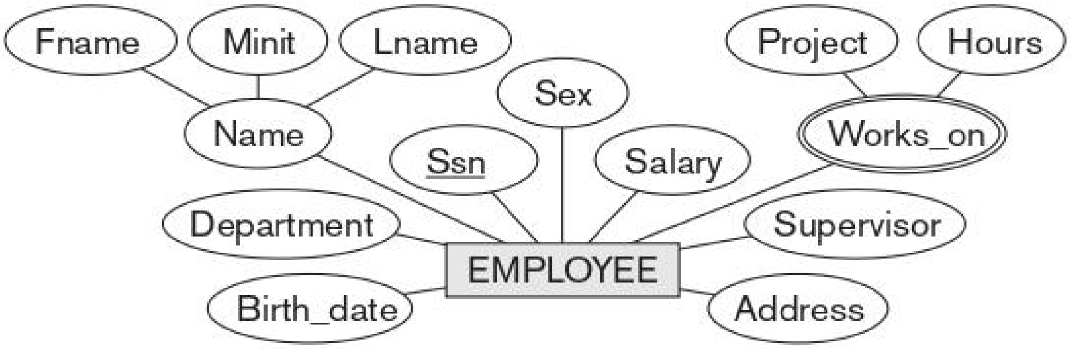
We can define the entity types for the COMPANY database. We can identify four entity types—one corresponding to each of the four items in the specification:

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute.
2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; the individual components of Name—First_name, Middle_initial, Last_name—or of Address.
4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

Figure: Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.






Fname
Minit
Lname
Project
Hours
Sex
Name
Ssn
Salary
Works_on
Department
Supervisor
EMPLOYEE
Birth_date
Address

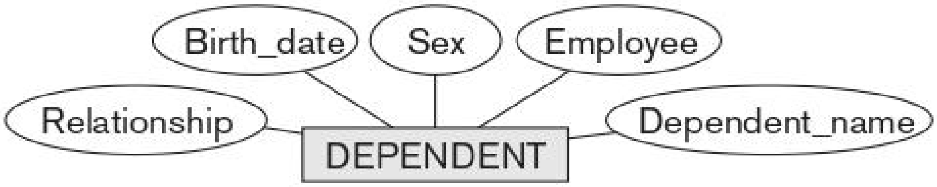
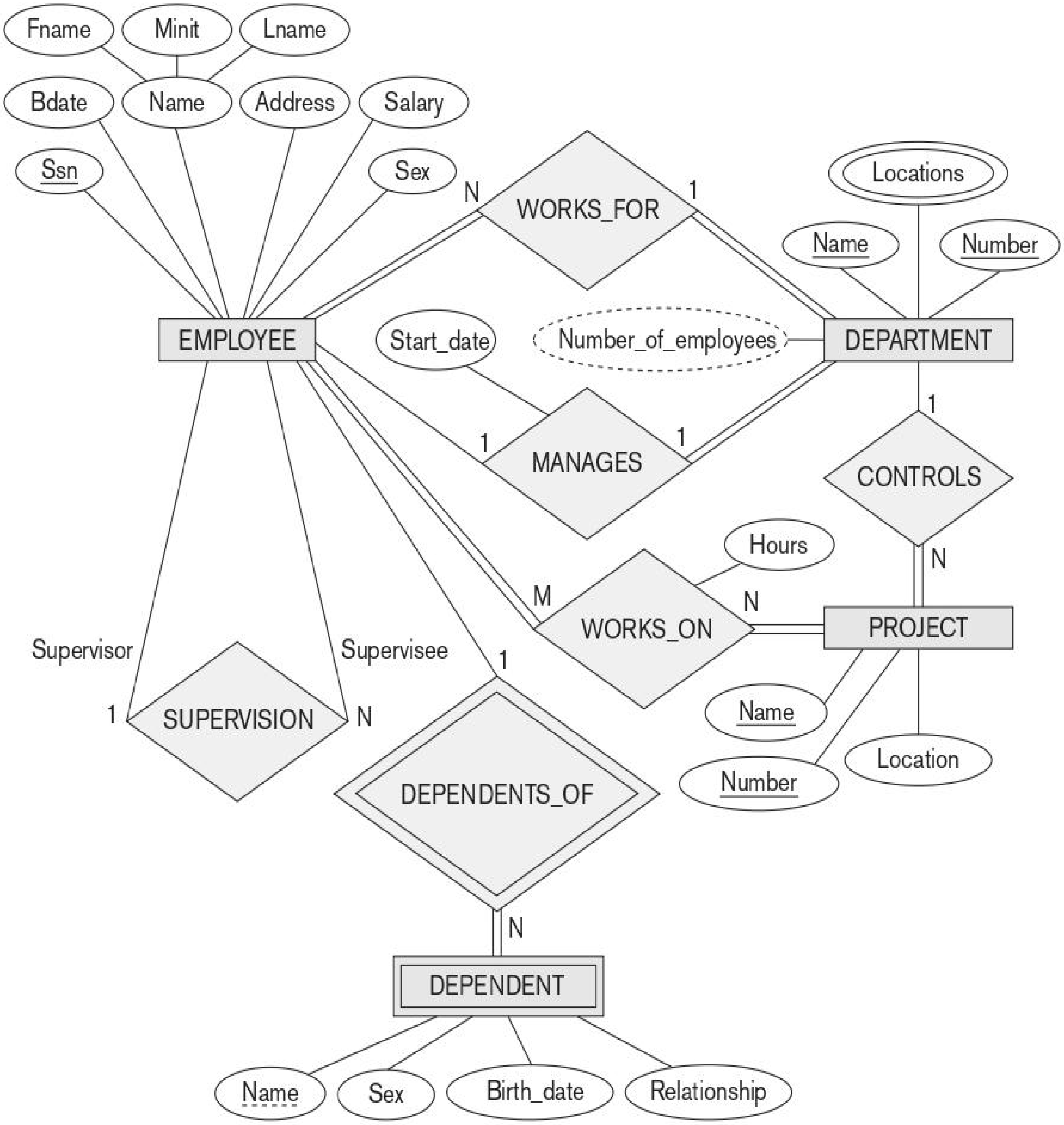

Birth_date
Sex
Employee
Relationship
Dependent_name
DEPENDENT

Figure A: An ER Schema diagram for the COMPANY database.



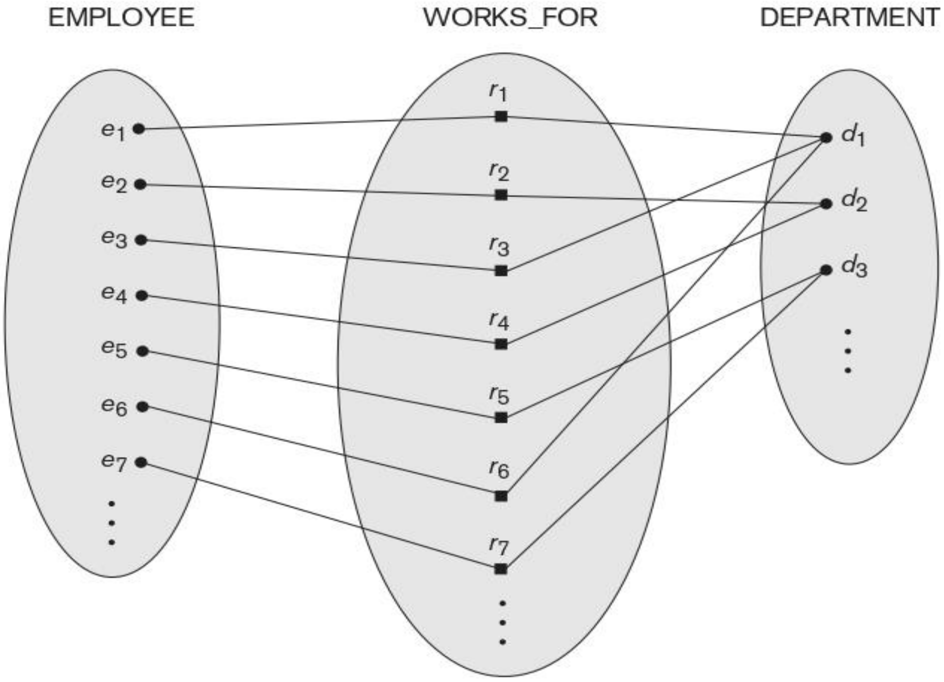
Relationship Types, Relationship Sets

In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists.

1. Relationship Types, Sets, and Instances:

A relationship type R among n entity types E1, E2, . . . , En defines a set of associations or a relationship set among entities from these entity types. As for the case of 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 $\mathbf{r_i}$, where each $\mathbf{r_i}$ associates **n** individual entities ($\mathbf{e_1, e_2, \ldots, e_n}$), and each entity **ej** in $\mathbf{r_i}$ is a member of entity set **Ej** , $1 \leq j \leq n$. Hence, a relationship set is a mathematical relation on E1, E2, . . . , En; alternatively, it can be defined as a subset of the Cartesian product of the entity sets E1 × E2 × . . . × En. Each of the entity types E1, E2, . . . , En is said to 'participate' in the relationship type R.

Example: Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT



A relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity.

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-shape box.

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 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 one of their attribute values. A weak entity type always has a **total participation constraint** with respect to its identifying relationship because a weak entity cannot be identified without an owner entity.

A weak entity type normally has a partial key, which is the attribute that can uniquely identify weak entities that are related to the same owner entity.

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 A). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes.

ER Diagrams

1. Summary of Notation for ER Diagrams:

In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful in database design.

Figure 'A' displays the COMPANY ER database schema as an ER diagram. Regular (strong) entity types such as EMPLOYEE, DEPARTMENT, and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the **Name** attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the **Locations** attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the **Number_of_employees** attribute of DEPARTMENT.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type.

In Figure A, the cardinality ratio of each binary relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT:EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for DEPARTMENT: EMPLOYEE in WORKS_FOR, and M:N for WORKS_ON.

2. Proper Naming of Schema Constructs

When designing a database schema, the choice of names for entity types, attributes, relationship types, and roles is not always straightforward. One should choose names that convey, as much as possible, the meanings attached to the different constructs in the schema. We choose to use **singular names** for entity types, rather than plural ones. In our ER diagrams, we will use the convention that entity type and relationship type names are in uppercase letters, attribute names have their initial letter capitalized, and role names are in lowercase letters.

As a general practice, the **nouns** appearing in the narrative tend to give rise to entity type names, and the **verbs** tend to indicate names of relationship types.

Another naming consideration involves choosing binary relationship names to make the ER diagram of the schema readable from left to right and from top to bottom.

3. Design Choices for ER Conceptual Design

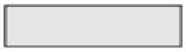
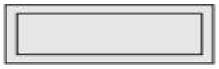
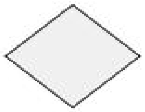
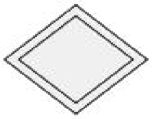
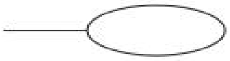
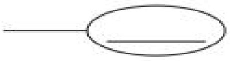
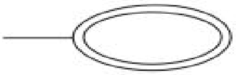
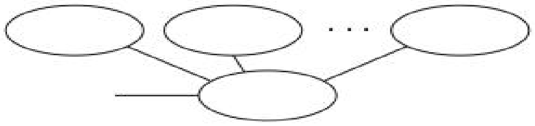
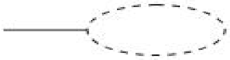
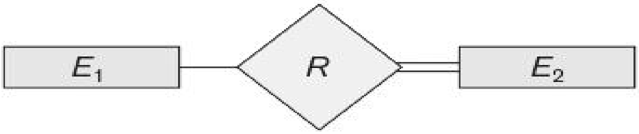
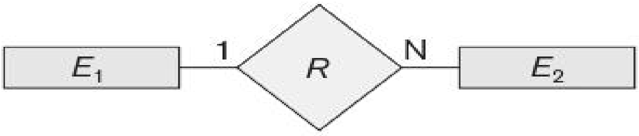
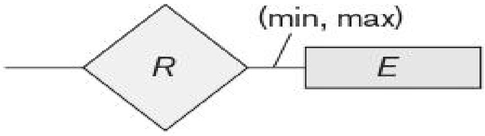
In general, the schema design process should be considered an iterative refinement process. Some of the refinements that are often used include the following:

- A concept may be first modeled as an attribute and then refined into a relationship.
- Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type.
- An inverse refinement to the previous case may be applied.

4. Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. The Unified Modeling Language (UML) notation for class diagrams, which has been proposed as a standard for conceptual object modeling.

Figure: Summary of the notation for ER diagrams

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Indentifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of E_2 in R
	Cardinality Ratio 1: N for E_1 : E_2 in R
	Structural Constraint (min, max) on Participation of E in R

The Relational Data Model and Relational Database Constraints

Relational Model Concepts

The relational model represents the database as a collection of relations. Each relation looks like a table of values or records. When a relation is considered as a table of values, each row in the table represents a collection of related data values or facts of the entity.

The table name and column names are used to interpret the meaning of the values in each row.

For example, in STUDENT table rows represents facts about a particular student entity. The column names are Name, Student_number, Class, and Major are used to interpret data values in each row. All values in a column are same data type.

In relational model terminology, a row is called a tuple, a column is called an attribute, and the table is called a relation. The data type describing the types of values that can appear in each column is represented by a domain of possible values.

Domains, Attributes, Tuples, and Relations

A domain 'D' is a set of atomic values. The atomic means each value in the domain is indivisible or not further divisible. A common method of specifying a domain is to specify a data type from which the data values forming the domain to specify a name for the domain or it helps in interpreting domain values.

Examples of domain:

Phone_numbers: The set of ten-digit phone numbers.

Social_security_numbers: The set of valid nine-digit Social Security numbers.

Names: The set of character strings that represent names of persons.

For example, the data type for the domain Phone_numbers can be declared as a ddd-ddd-dddd, where each 'd' is a numeric digit.

Relation schema:

A relation schema2 R, denoted by R(A1, A2, … , An), is made up of a relation name 'R' and a list of attributes, A1, A2, … , An. Each attribute Ai is the name of some domain 'D' in the relation schema R. 'D' is called the domain of Ai and is denoted by dom(Ai). The degree of a relation is the number of attributes 'n' of its relation schema.

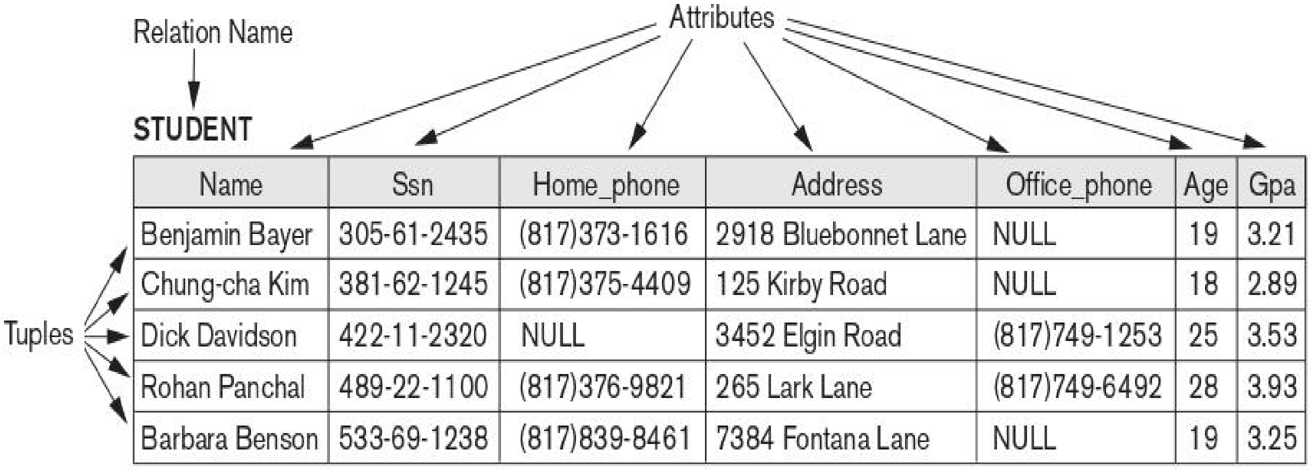
For ex, the relation sechema STUDENT of degree seven is

STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

Relation:

A relation 'r' of the relation schema R(A1, A2, … , An), denoted by r(R), is a set of n-tuples r = {t1, t2, … , tm}. Each n-tuple 't' is an ordered list of 'n' values. t = < V1, V2,……Vn> where each value vi , 1 ≤ i ≤ n, is an element of dom (Ai) or is a special NULL value. The ith value in tuple t, which corresponds to the attribute Ai , is referred to as t[Ai]. For example, the STUDENT relation which corresponds to the STUDENT schema.

The attributes and tuples of relation STUDENT


Relation Name
Attributes
STUDENT
Tuples

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

Characteristics of Relations

The characteristics of a relation are

1. **Ordering of Tuples in a Relation:** A relation is defined as a set of tuples. Tuples in a relation do not have any particular order. When we display a relation as a table, the rows are displayed in a certain order.

2. **Ordering of Values within a Tuple:** An n-tuple is an ordered list of 'n' values, so the ordering of values in a tuple is important.

3. **Values and NULLs in the Tuples:** Each value in a tuple is an atomic value; that is, it is not divisible into components in the relational model. Composite and multivalued attributes are not allowed.

The NULL values, which are used to represent the values of attributes are unknown. For example, STUDENT has a NULL value for home_phone because does not have a home_phone.

Relational Model Notation

The following notations are used in relational model.

1. A relation schema R of degree n is denoted by R(A1, A2, … , An)
2. The uppercase letters Q, R, S denote relation names.
3. The lowercase letters q, r, s denote relation states.
4. The letters t, u, v denote tuples.
5. An attribute 'A' can be qualified with the relation name 'R' to which it belongs by using the dot notation R.A. For example, STUDENT.Name or STUDENT.Age.
6. An n-tuple 't' in a relation r(R) is denoted by t = <V1,V2,……Vn> , where Vi is the value corresponding to attribute Ai.

The following notation refers to component values of tuples:

Both t[Ai] and t.Ai refers to the value Vi in t for attribute Ai.

Relational Model Constraints and Relational Database Schemas

There are many restrictions or constraints on the actual values in the database. These constraints are derived from the rules in the world.

Constraints on databases can generally be divided into three main categories:

1. Constraints are inherent in the data model. These constraints are called **implicit constraints**.
2. Constraints are directly expressed in schemas of the data model. These constraints or **explicit constraints**.
3. Constraints are not directly expressed in the schemas of the data model, and hence must be expressed by the application programs. These constraints are called **application-based constraints**.

The relational model we mainly used schema based constraints. The schema based constraints are:

a) Domain constraints,
b) Key constraints and constraints on NULL values.
c) Entity integrity constraints, and
d) Referential integrity constraints.

1. Domain Constraints: Domain constraints specify that within each tuple, the value of each attribute 'A' must be an atomic value from the domain dom(A).

2. Key Constraints and Constraints on NULL Values: A relation is defined as a set of tuples. The elements of a set are distinct. Any two tuples cannot have the same combination of values for their attributes. Usually, other subset of attributes in the relation schema have the same combination of values.

Super Key: Suppose a subset of attributes by SK, then any two distinct tuples t1 and t2 in a relation we have the constraint that:

$$t1[SK] \neq t2[SK]$$

Any such set of attributes SK is called a **super key** of the relation schema R. A superkey SK specifies a uniqueness constraints.

Key: A superkey can have redundant attributes, a key has no redundancy attributes.

A key satisfies two properties:

1. Two distinct tuples in the relation cannot have identical values for the attributes in the key.
2. It is a minimal superkey from which we cannot remove any attribute.

For example, Consider the STUDENT relation. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.

The super key {Ssn, Name, Age} is a superkey. The superkey {Ssn, Name, Age} is not a key of STUDENT. Any superkey formed from a single attribute is also a key.

Candidate Key: In general, a relation schema may have more than one key. Each of the keys is called a **candidate key**.

For example, the CAR relation has two candidate keys: License_number and Engine_serial_number.

It is common to designate one of the candidate keys as the primary key of the relation.

NOT NULL: Another constraint on attributes specifies NULL values are permitted or NULL values are not permitted.

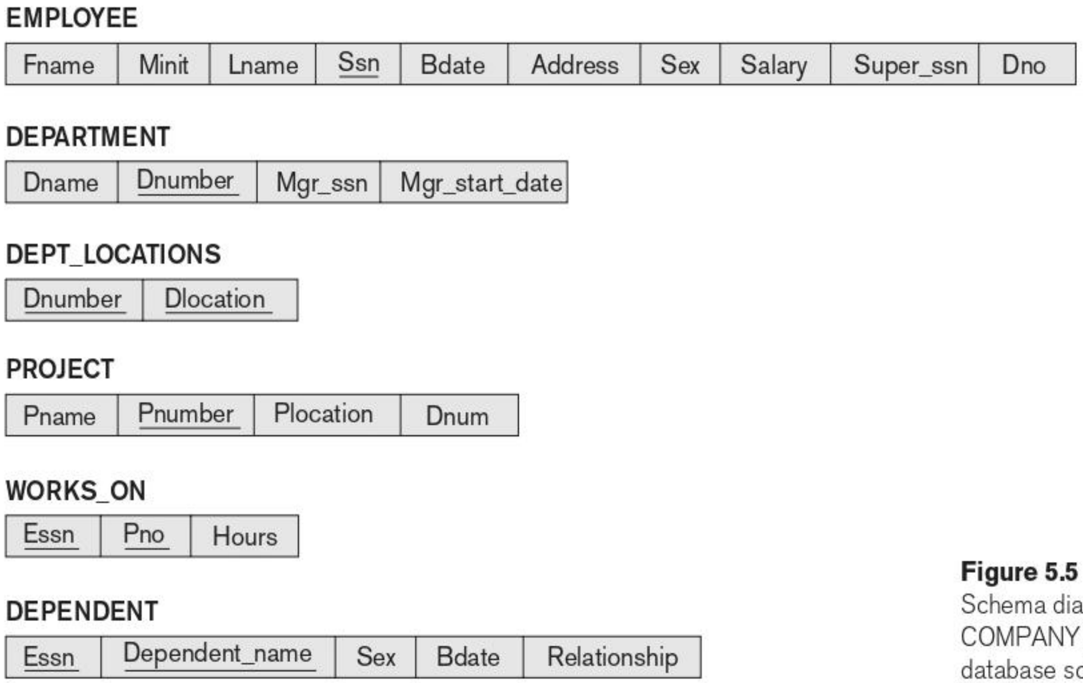
For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then 'Name' of STUDENT is constrained to be NOT NULL.

3. Relational Databases and Relational Database Schemas

A relational database contains many relations, with tuples in relations that are related in various ways.

A **relational database schema 'S'** is a set of relation schemas S = {R1, R2, … , Rm} and a set of integrity constraints IC.

The following Figure shows a relational database schema
COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}.
The underlined attribute represents the primary key.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure 5.5
Schema diagram for the COMPANY relational database schema.

A database state that does not obey all the integrity constraints is called not valid or **invalid state** and a state that satisfies all the constraints is called a **valid state.**

4. Entity Integrity, Referential Integrity, and Foreign Keys:

The **entity integrity constraint** states that no primary key value can be NULL. Because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key we cannot identify tuples.

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations.

For example, the attribute Dno of EMPLOYEE gives the department number must match the Dnumber value of some tuple in the DEPARTMENT relation.

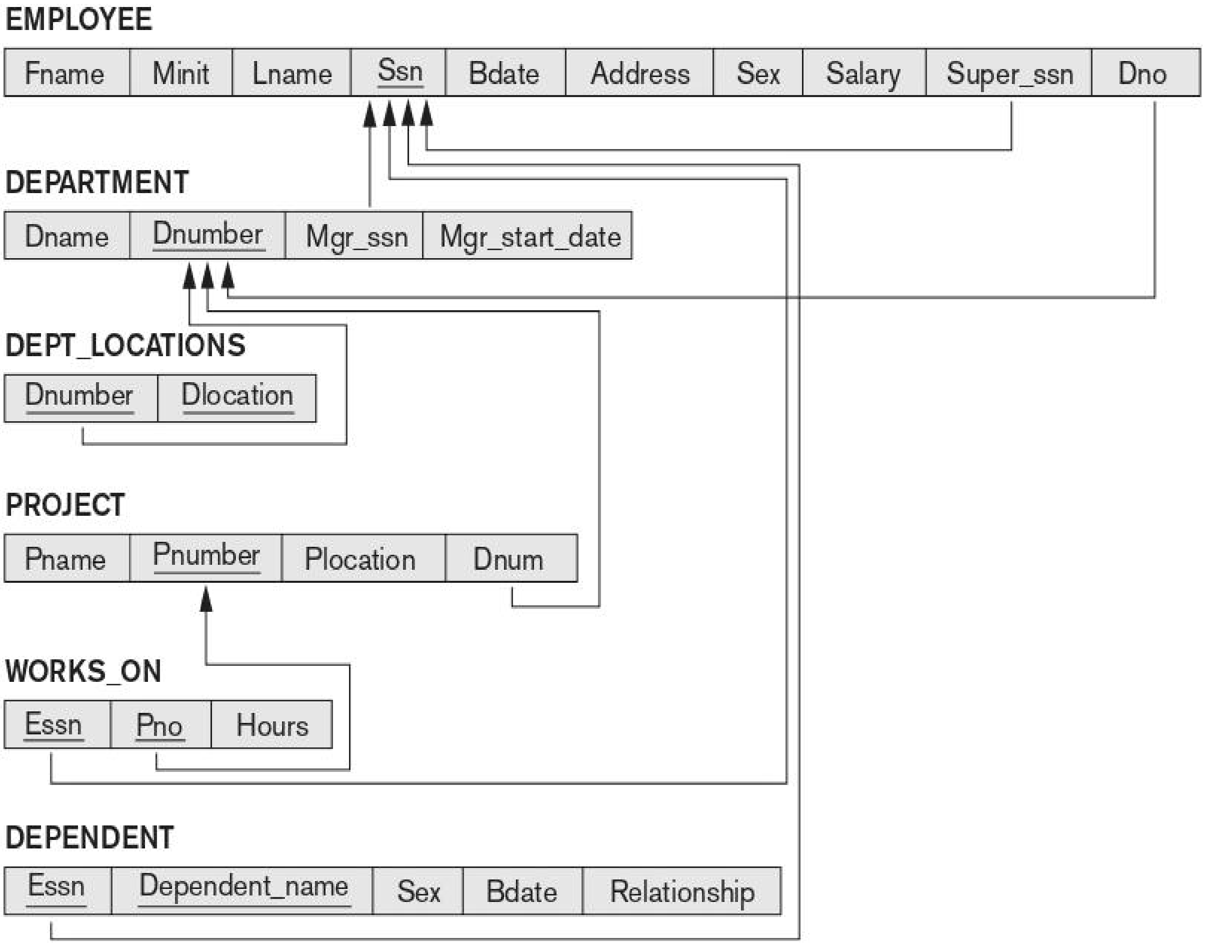
To define referential integrity more formally, we define the concept of a **foreign key.** The concept of foreign key is to specify a referential integrity constraints between the two relation schemas R1 and R2.

A set of attributes FK in relation schema R1 is a foreign key of R1 that references relation R2 if it satisfies the following rules:

1. The attributes in FK as the primary key attributes PK of R2; the attributes FK are said to reference relation R2.

2. A value of FK in a tuple t1 as a value of PK for some tuple t2 or is NULL.
We have t1[FK] = t2[PK], and we say that the tuple t1 references to the tuple t2.

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



Update Operations, Transactions, and Dealing with Constraint Violations

There are three basic operations in relations. They are Insert, Delete, and Update (or Modify).

Insert is used to insert one or more new tuples in a relation.
Delete is used to delete tuples from relations.
Update is used to change the values of some attributes in existing tuples.

Those operations are applied the integrity constraints specified on the relational database schema should not be violated.

Insert Operation:

The Insert operation provides a list of attribute values for a new tuple **t** that is to be inserted into a relation **R**. Insert can violate any of the four types of constraints.

1. **Domain constraints:** Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain.
2. **Key constraints:** Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R).
3. **Entity integrity constraints:** These can be violated if any primary key of the new tuple **t** is NULL.
4. **Referential integrity constraints:** These can be violated if the value of any foreign key refers to a tuple that does not exist in the referenced relation.

Delete Operation:

The Delete operation can violate only referential integrity. If the tuple being deleted is referenced by foreign keys from other tuples in the database.

Update Operation: The Update (or Modify) operation is used to change the values of one or more attributes in a tuple of some relation **R**. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified.

The Transaction Concept

A database application program running against a relational database typically executes one or more transactions. A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints. A single transaction may involve any number of retrieval operations.

A large number of commercial applications running against relational databases in **online transaction processing** (OLTP) systems are executing transactions at rates that reach several hundred per second.

Relational Database Design Using ER to Relational Mapping

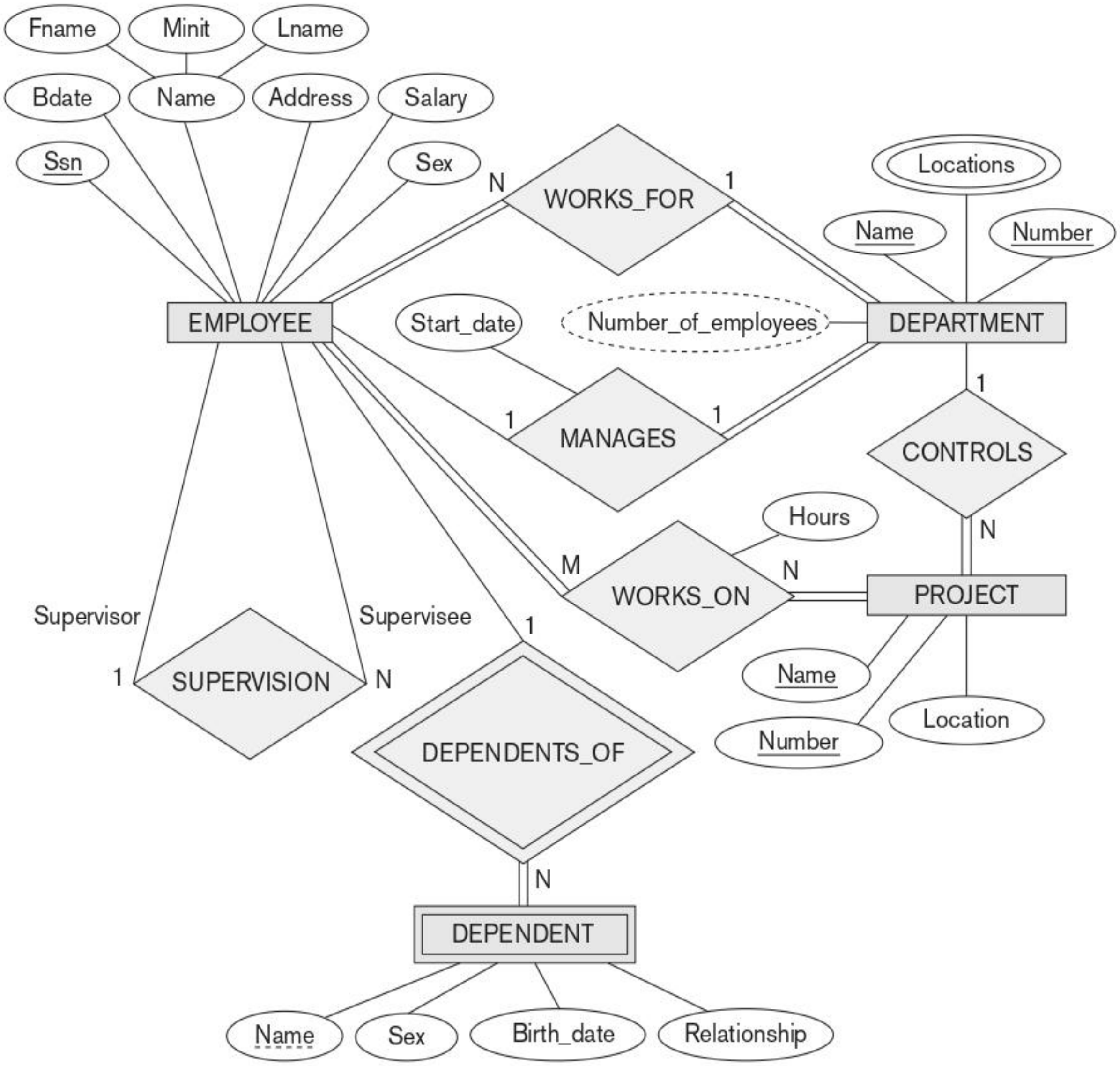
In this chapter we are learning how to design a relational database schema based on a conceptual schema design. This corresponds to the **logical database design** or **data model mapping.**

We have used seven algorithms to convert the basic ER model construct entity types (strong and weak), binary relationships (with various structural constraints), n-ary relationships and attributes (simple, composite and multivalued) into relations. We have also used some other algorithms how to map Specialization/Generalization and union types into relations.

ER to Relational Mapping Algorithms:

We describe the steps of an algorithm for ER-to-relational mapping. The ER diagram for COMPANY schema is converted into relational database schema using seven algorithms.

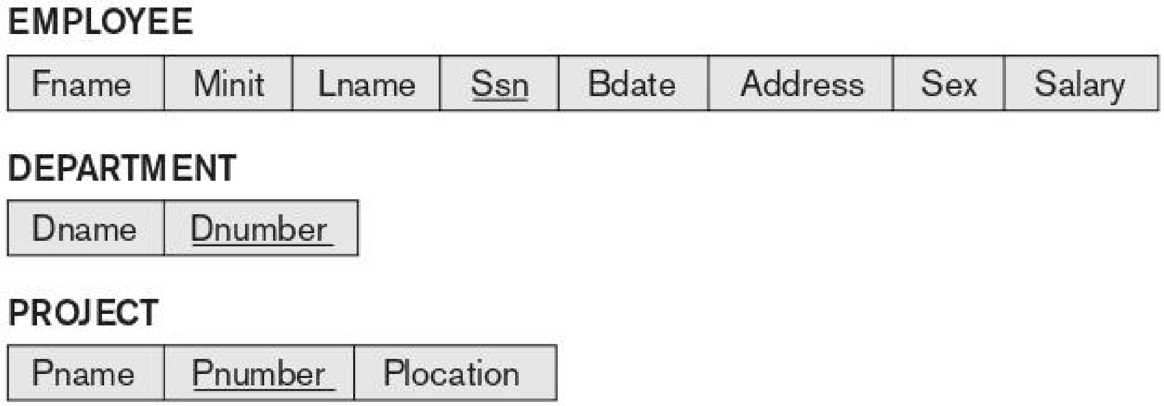
An ER Schema diagram for the COMPANY database.



Step 1: Mapping of Regular Entity Types:

For each regular (strong) entity type E in the ER schema, create a relation R of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT. We choose **Ssn, Dnumber,** and **Pnumber** as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT respectively.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary

DEPARTMENT

Dname	Dnumber

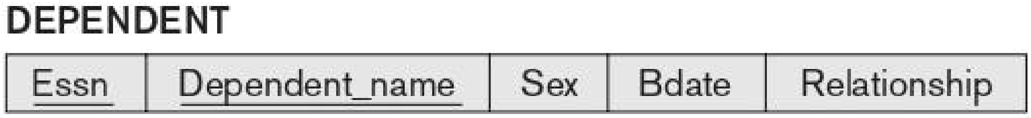
PROJECT

Pname	Pnumber	Plocation

Step 2: Mapping of Weak Entity Types:

For each **weak entity type W** in the ER schema with owner entity type E, create a relation R and include all simple of W as attributes of R. In addition, include as **foreign key** attributes of R, the **primary key** attribute of the relation that correspond to the **owner entity.** The **primary key** of R is the combination of the primary key of the owner and the partial key of the weak entity W.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. We include the primary key **Ssn** of the EMPLOYEE relation which corresponds to the owner entity type as a foreign key attribute of DEPENDENT. The primary key of the DEPENDENT relation is the combination {Essn, Dependent_name}.



DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Step 3: Mapping of Binary 1:1 Relationship Types:

For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the crossreference or relationship relation approach. The first approach is the most useful.

1. **Foreign key approach:** Choose one of the relations and S, include as a foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S.

In our example, choose the DEPARTMENT relation because its participate totally on the MANAGES relationship, we include primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it to Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship in the DEPARTMENT relation and rename it Mgr_start_date.

DEPARTMENT

Dname	**<u>Dnumber</u>**	**Mgr_ssn**	**Mgr_start_date**

2. **Merged relation approach:** An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when both participations are total, as this would indicate that the two tables will have the exact same number of tuples at all times.
3. **Cross-reference or relationship relation approach:** The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation R is called a relationship relation (or sometimes a lookup table).

Step 4: Mapping of Binary 1:N Relationship Types:

For each regular binary 1:N relationship type R, identify the relation S that participating entity type at the N-side of the relationship type. Include as foreign key in S the primary key of the relation T that participating in 'R'.

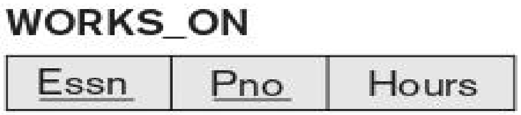
In our example, we map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION.

1. For WORKS_FOR, we include the primary key **Dnumber** of the DEPARTMENT relation as foreign key in the EMPLOYEE relation.
2. For SUPERVISION, we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself.
3. For CONTROLS, we include the primary key **Dnumber** of DEPARTMENT relation as foreign key in the PROJECT relation.

Step 5: Mapping of Binary M:N Relationship Types:

For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship type as attributes of S.

In our example, we map the M:N relationship type WORKS_ON by creating the relation WORKS_ON. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them **Pno** and **Essn.**



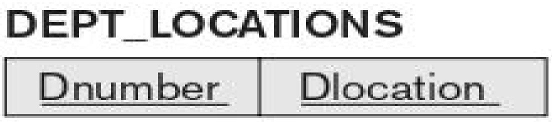
WORKS_ON

Essn	Pno	Hours

Step 6: Mapping of Multivalued Attributes:

For each multivalued attribute **A**, create a new relation R. This relation R will include an attribute **A**, plus the primary key attribute **K** as a foreign key in R of the relation that represents the entity type.

In our example, we create a relation DEPT_LOCATIONS. The attribute **Dlocation** represents the multivalued attribute LOCATIONS of DEPARTMENT, whereas **Dnumber** as foreign key represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of **{Dnumber, Dlocation}.**



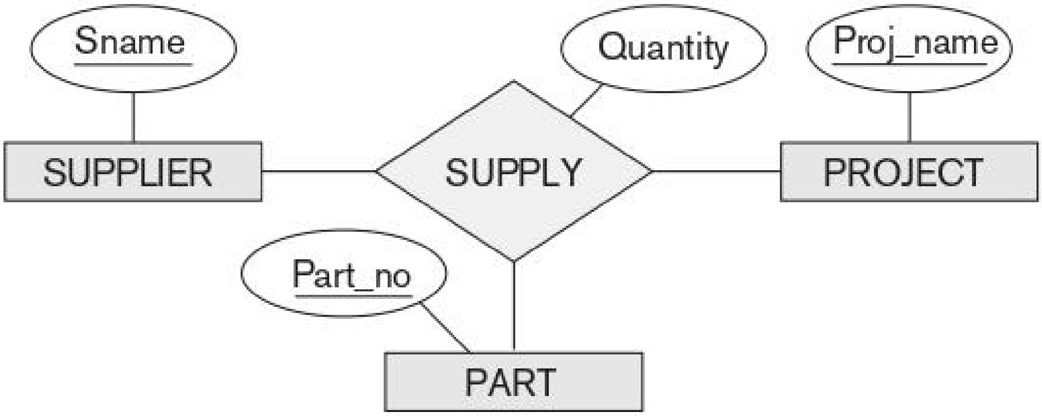
DEPT_LOCATIONS

Dnumber	Dlocation

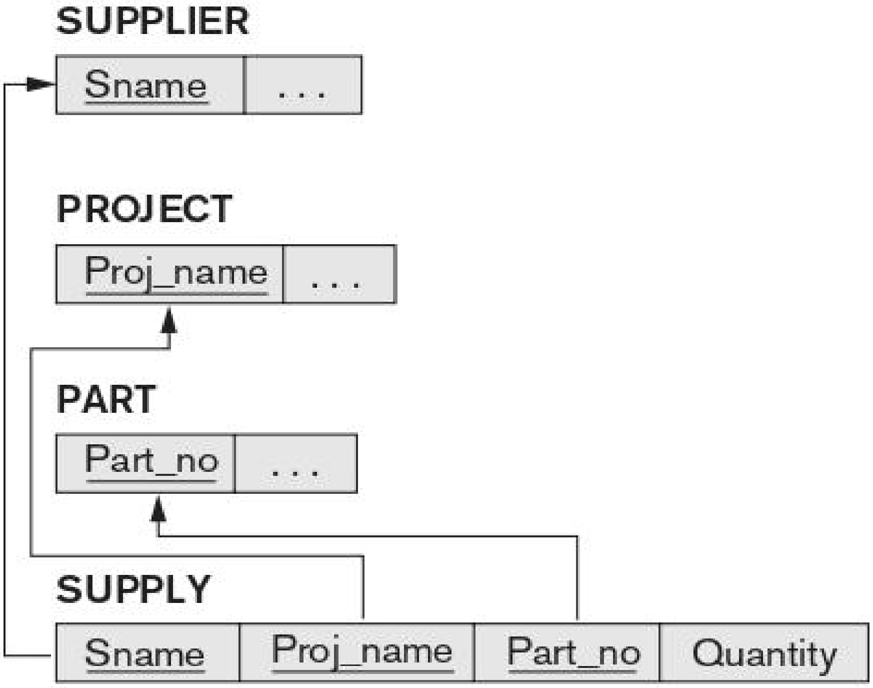
Step 7: Mapping of N-ary Relationship Types:

For each n-ary relationship type R, where n > 2, create a new relationship relation S to represent R. Include as foreign key attributes in S. Primary keys of the relations also include any simple attributes of the n-ary relationship type as attributes of S.

Consider the relationship type SUPPLY as shown below.



This can be mapped to the relation SUPPLY as shown below.



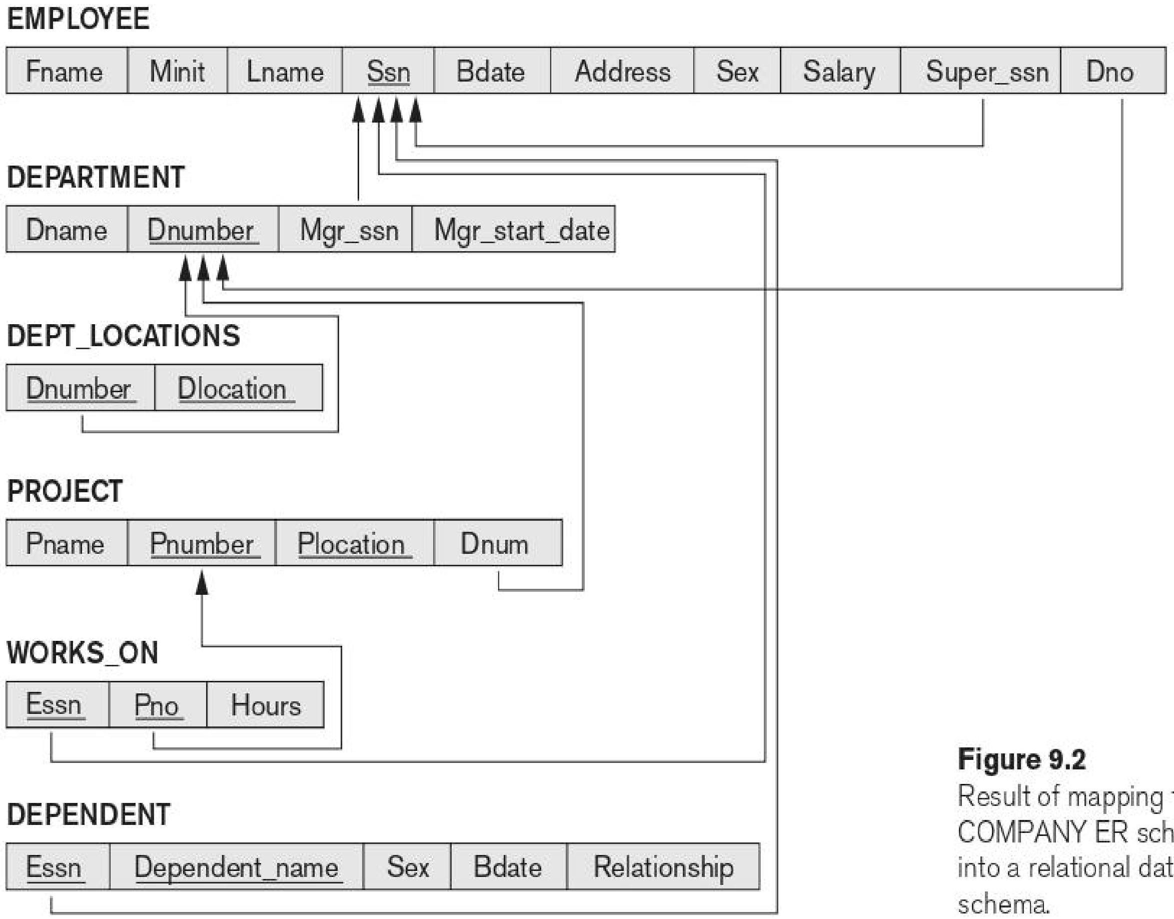


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

****

UNIT - II

Relational Databases Design

As with many design problems, database design may be performed using two approaches: **bottom-up or top-down**. A **bottom-up** design methodology (also called design by synthesis) considers the basic relationships among individual attributes as the starting point and uses those to construct relation schemas. This approach is not very popular in practice. In contrast, a **top-down** design methodology (also called design by analysis) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report.

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are information preservation and minimum redundancy. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships.

Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, wwe discuss four informal guidelines that may be used as measures to determine the quality of relation schema design:

1. Making sure that the semantics of the attributes is clear in the schema
2. Reducing the redundant information in tuples
3. Reducing the NULL values in tuples
4. Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another.

1.Imparting Clear Semantics to Attributes in Relation:

Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them. The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple.

The **semantics** of the relation is a relation exactly means and stands for—the better the relation schema design. To illustrate this, consider Figure 1, a simplified version of the COMPANY relational database schema, and Figure 2, which presents an example of populated relation states of this schema.

The meaning of the EMPLOYEE relation schema is simple: Each tuple represents an employee, with values for the employee's name (Ename), Social Security number (Ssn), birth date (Bdate), and address (Address), and the number of the department that the employee works for (Dnumber). The Dnumber attribute is a foreign key that represents an implicit relationship between EMPLOYEE and DEPARTMENT.

The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute Dmgr_ssn of DEPARTMENT relates a department to the employee who is its manager, whereas Dnum of PROJECT relates a project to its controlling department; both are foreign key attributes.

Figure: 1 A simplified COMPANY relational database schema.

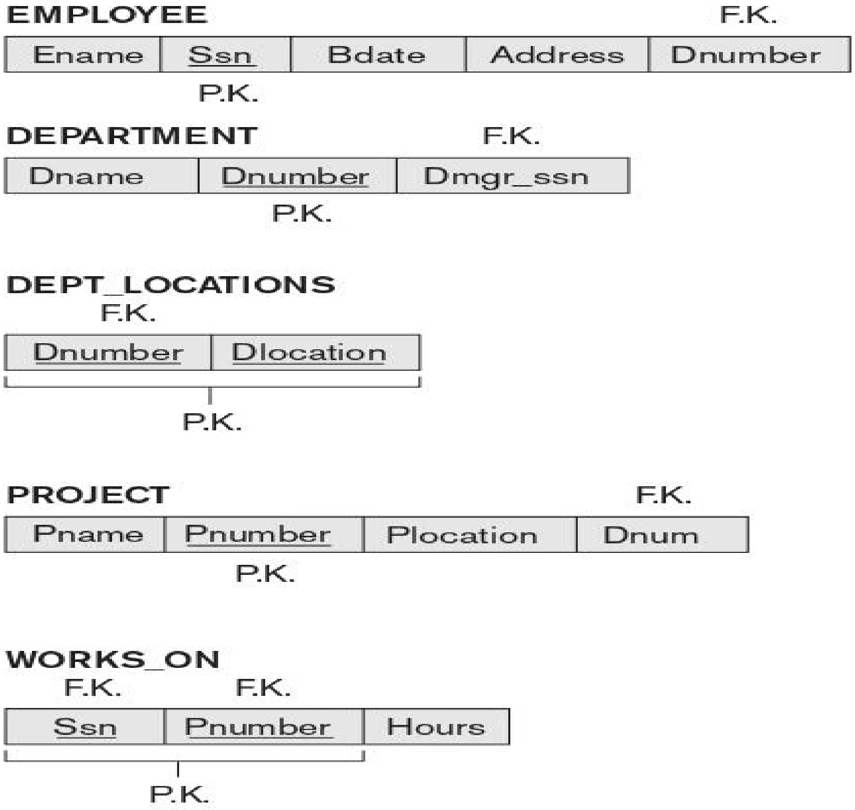
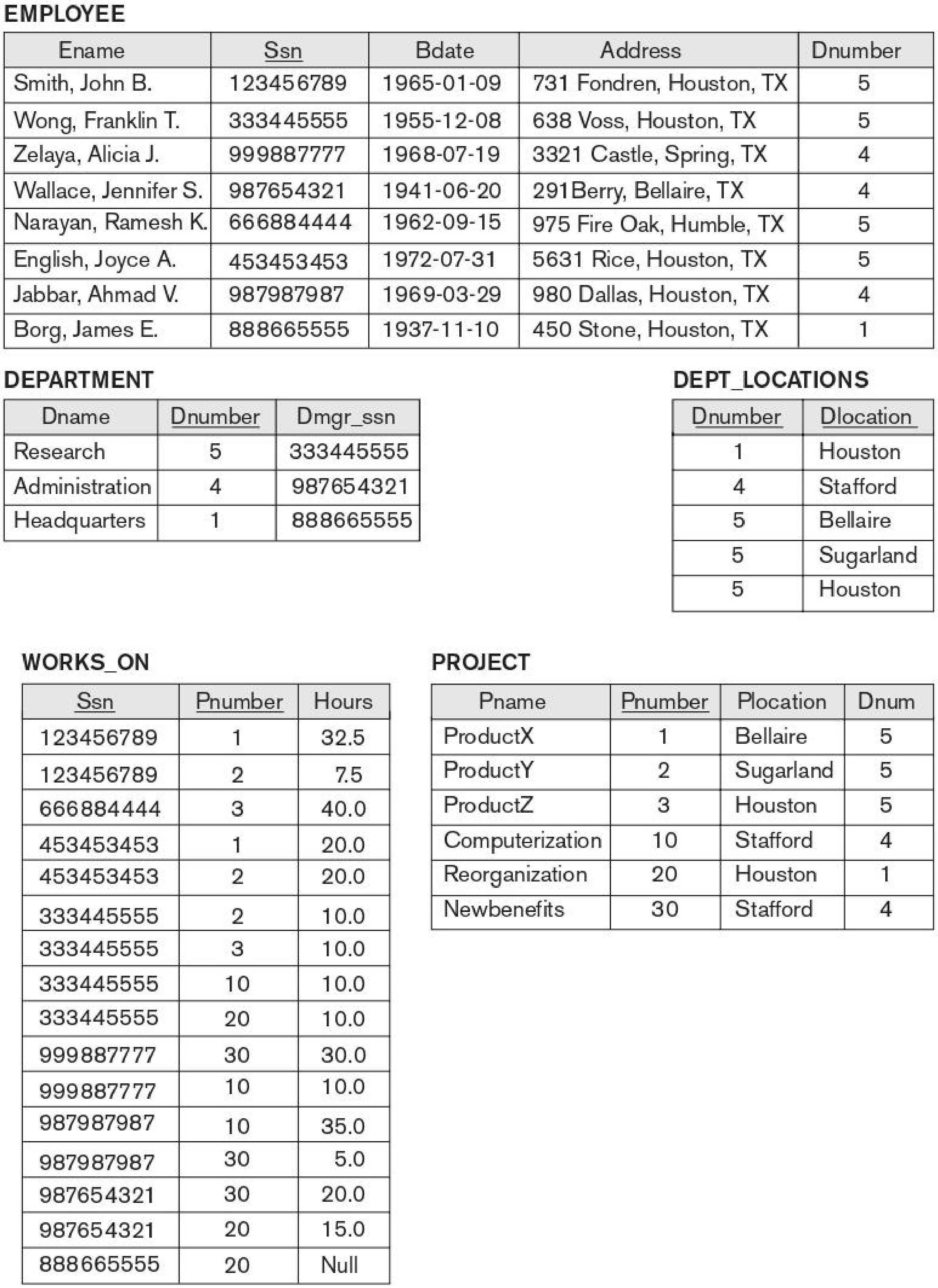


Figure: 2 Sample database state for the relational database schema in Figure 1



EMPLOYEE

Ename	Ssn	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

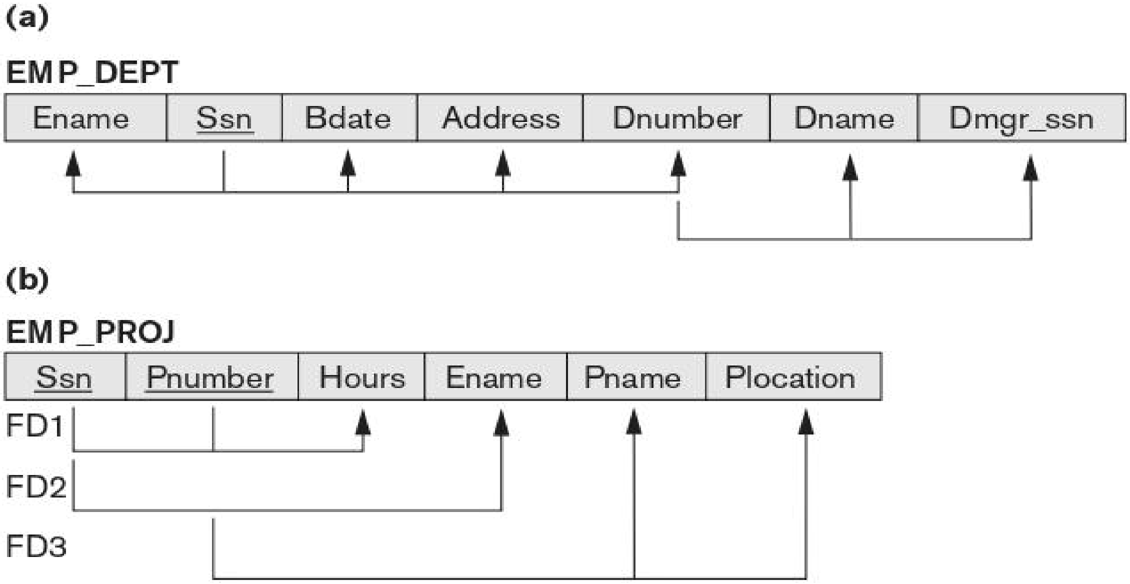
Ssn	Pnumber	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	Null

PROJECT

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

Guideline 1: Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Guideline 1: The relation schemas in the foloowing Figures (a) and (b) also have clear semantics. A tuple in the EMP_DEPT relation schema in Figure (a) represents a single employee but includes, along with the Dnumber, additional information—namely, the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssn) of the department manager. For the EMP_PROJ relation in Figure (b), each tuple relates an employee to a project but also includes the employee name (Ename), project name (Pname), and project location (Plocation).

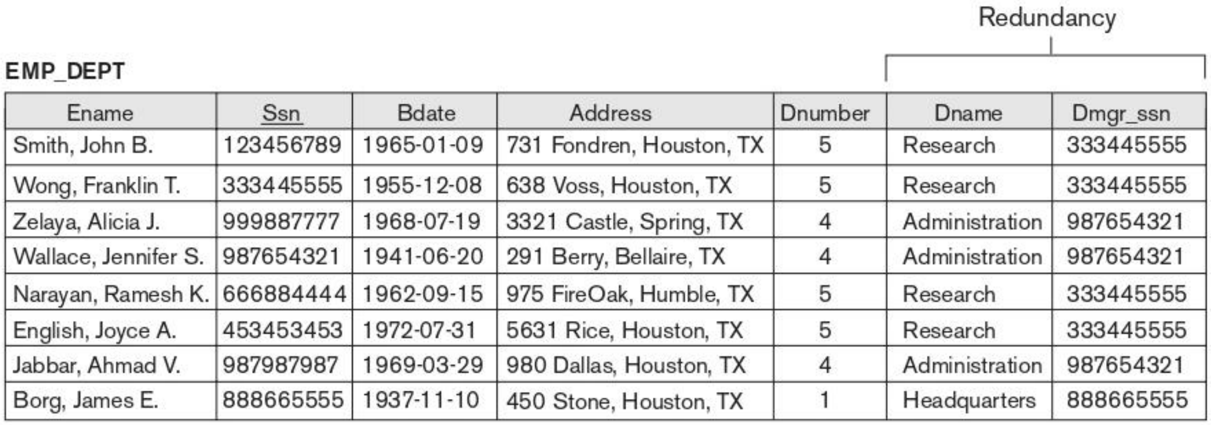


2. Redundant Information in Tuples and Update Anomalies:

One goal of schema design is to minimize the storage space used by the base relation. Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 2.

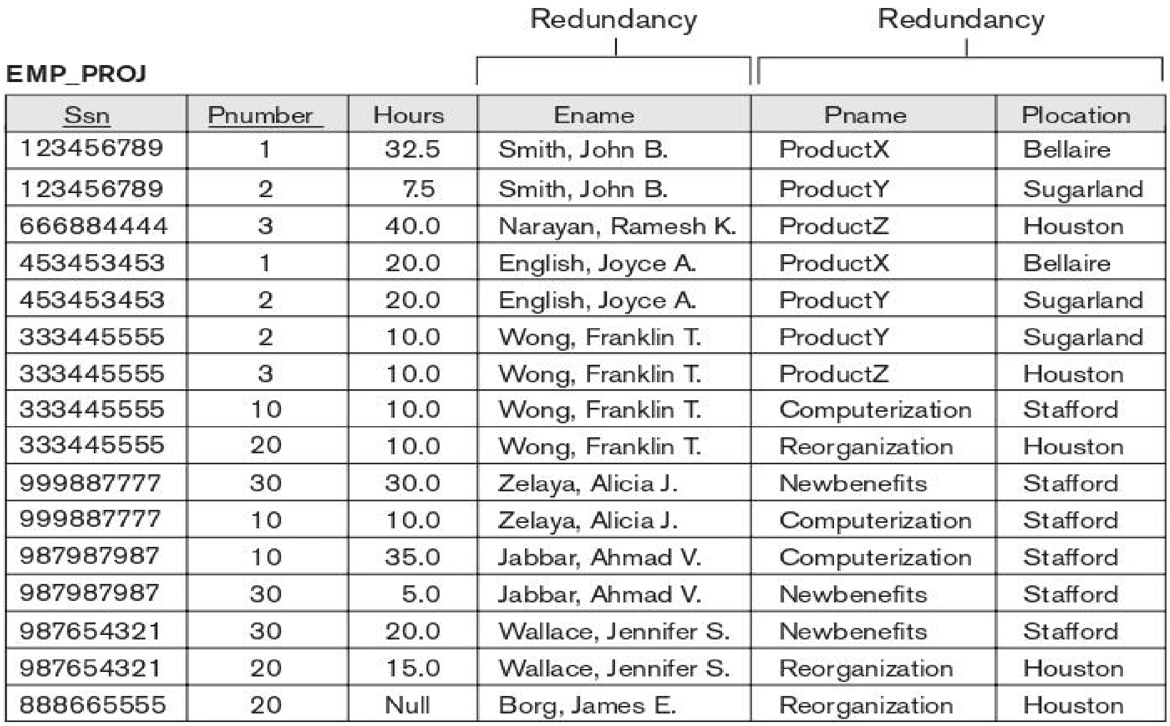
In Figure 3, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation (see Figure 4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

Figure 3: Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 2


Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555


Redundancy
Redundancy

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation
123456789	1	32.5	Smith, John B.	ProductX	Bellaire
123456789	2	7.5	Smith, John B.	ProductY	Sugarland
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston
888665555	20	Null	Borg, James E.	Reorganization	Houston

Storing natural joins of base relations leads to an additional problem referred to as update anomalies. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.

Insertion Anomalies: Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

1. To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs. For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are consistent with the corresponding values for department 5 in other tuples in EMP_DEPT

2. It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because its primary key Ssn cannot be null.

Deletion Anomalies: If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost inadvertently from the database. This problem does not occur in the database of Figure 2 because DEPARTMENT tuples are stored separately.

Modification Anomalies: In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent.

Guideline 2: Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

3. NULL Values in Tuples: In some schema designs we may group many attributes together into a “fat” relation. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level. Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:

1. The attribute ***does not apply*** to this tuple. For example, Visa_status may not apply to U.S. students.
2. The attribute value for this tuple is ***unknown***. For example, the Date_of_birth may be unknown for an employee.
3. The value is ***known but absent***; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

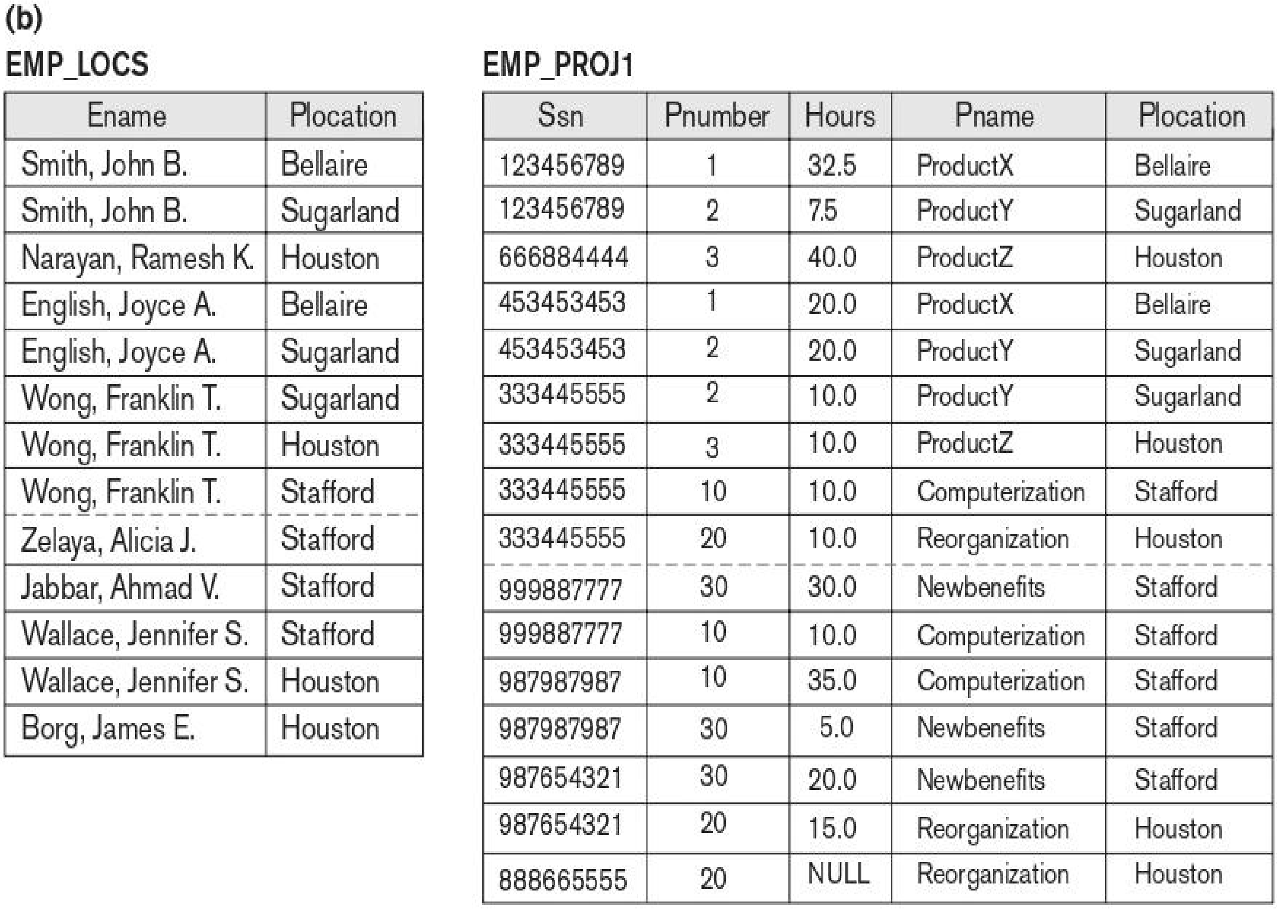
Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we state another guideline.

Guideline 3: As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

4. Generation of Spurious Tuples: Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS. If we attempt a NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ.

In Figure 4, the result of applying the join to only the tuples for employee with Ssn = “123456789” is shown. Additional tuples that were not in EMP_PROJ are called **spurious tuples** because they represent spurious information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 4. It is left to the reader to complete the result of NATURAL JOIN operation on the EMP_PROJ1 and EMP_LOCS tables in their entirety and to mark the spurious tuples in this result.

Figure: The result of projecting the extension of EMP_PROJ from Figure 3 onto the relations EMP_LOCS and EMP_PROJ1.



(b)

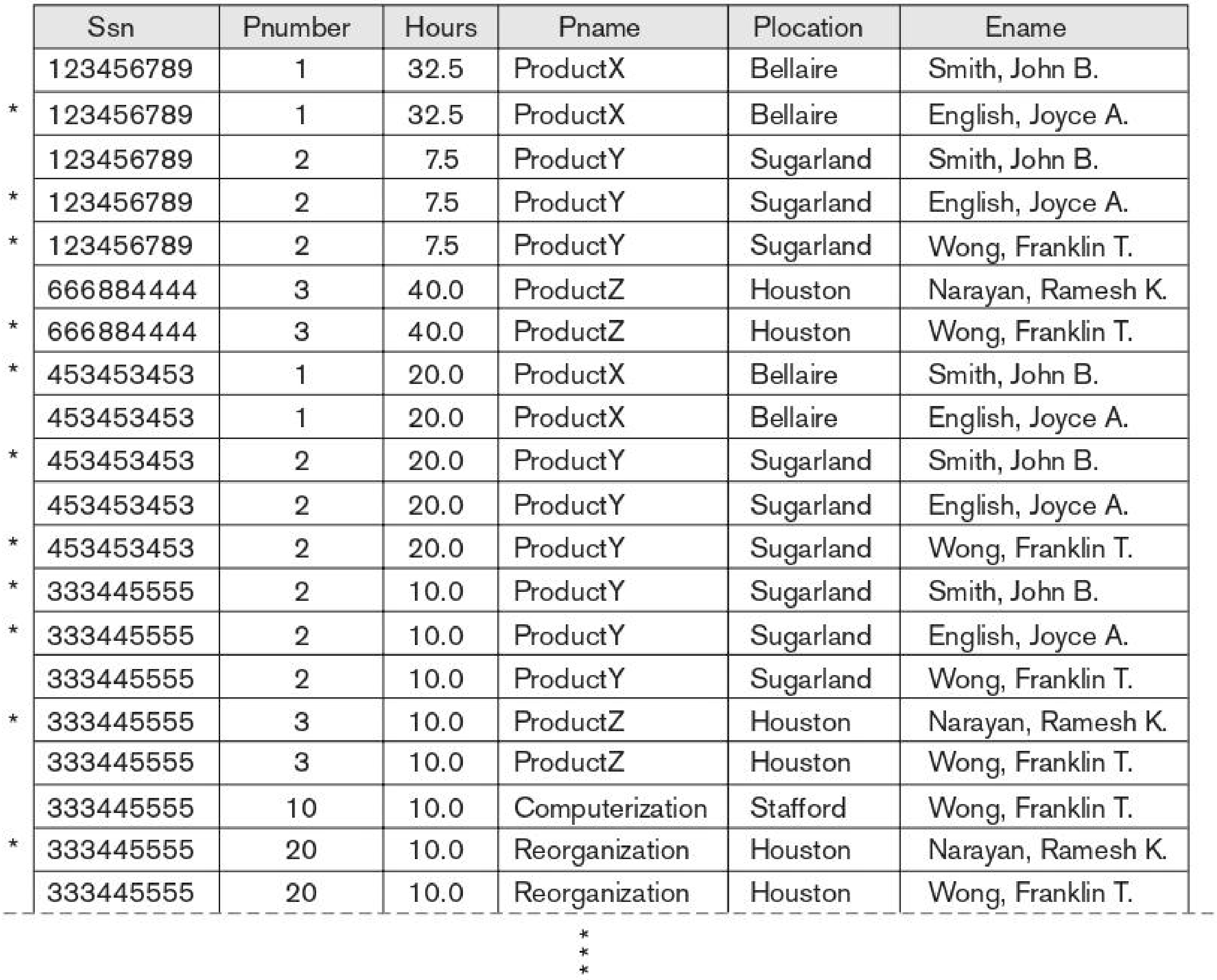
EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

Figure 4: Result of applying NATURAL JOIN to the tuples in EMP_PROJ1 and EMP_LOCS of above Figure just for employee with Ssn = “123456789”. Generated spurious tuples are marked by asterisks.



	Ssn	Pnumber	Hours	Pname	Plocation	Ename
	123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
	123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
	666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	ProductZ	Houston	Wong, Franklin T.
*	453453453	1	20.0	ProductX	Bellaire	Smith, John B.
	453453453	1	20.0	ProductX	Bellaire	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Smith, John B.
	453453453	2	20.0	ProductY	Sugarland	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	2	10.0	ProductY	Sugarland	Smith, John B.
*	333445555	2	10.0	ProductY	Sugarland	English, Joyce A.
	333445555	2	10.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	3	10.0	ProductZ	Houston	Narayan, Ramesh K.
	333445555	3	10.0	ProductZ	Houston	Wong, Franklin T.
	333445555	10	10.0	Computerization	Stafford	Wong, Franklin T.
*	333445555	20	10.0	Reorganization	Houston	Narayan, Ramesh K.
	333445555	20	10.0	Reorganization	Houston	Wong, Franklin T.

Guideline 4: Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated. Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

Functional Dependencies

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes A1, A2, … , An; let us think of the whole database as being described by a single universal relation schema R = {A1, A2, … , An}.

Definition: A functional dependency, denoted by X → Y, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t1 and t2 in r that have t1[X] = t2[X], they must also have t1[Y] = t2[Y].

1. This means that the values of the Y component of a tuple in 'r' depends on, the values of the X component.
2. Alternatively, the values of the X component determine the values of the Y component. We also say that the functional dependency from X to Y, or that Y is functionally dependent on X. The abbreviation for functional dependency is FD or f.d. The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of r(R) agree on their X-value, they must necessarily agree on their Y-value.

1. X is a **candidate key** of R—this implies that X → Y for any subset of attributes Y of R.
2. If X → Y in R, this does not say whether or **not** Y → X in R.

Consider the relation schema EMP_PROJ:

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation

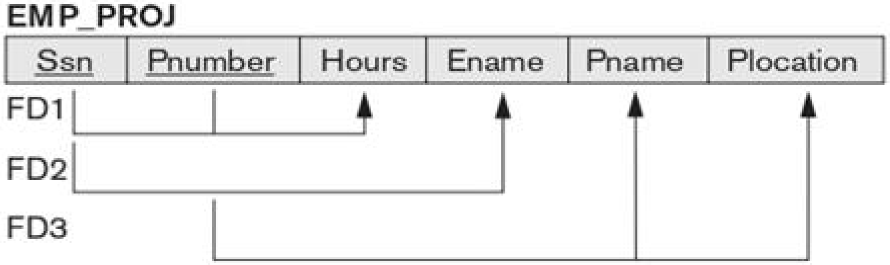
The functional dependencies of the above rational schema is

a. Ssn → Ename
b. Pnumber → {Pname, Plocation}
c. {Ssn, Pnumber} → Hours

These functional dependencies specify that

(a) The value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename),
(b) The value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and
(c) A combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

The following figure shows the diagrammatic notation of FD.



Each FD is displayed as a **horizontal line**. The **left hand** side attributes of the FD are connected by a **vertical lines** to the line representing the FD, while the **right hand** side attributes are connected by arrows pointing towards the attributes.

Inference Rules for Funtional Dependencies:

A functional dependency is a property of the relation schema R, not of a particular legal relation state r of R. Therefore, an FD cannot be inferred automatically from a given relation extension **r** but must be defined explicitly by someone who knows the semantics of the attributes of R.

For example, the following Figure shows a particular state of the TEACH relation schema. Although at first glance we may think that Text → Course, we cannot confirm this unless we know that it is true for all possible legal states of TEACH. For example, because 'Smith' teaches both 'Data Structures' and 'Database Systems,' we can conclude that Teacher does not functionally determine Course.

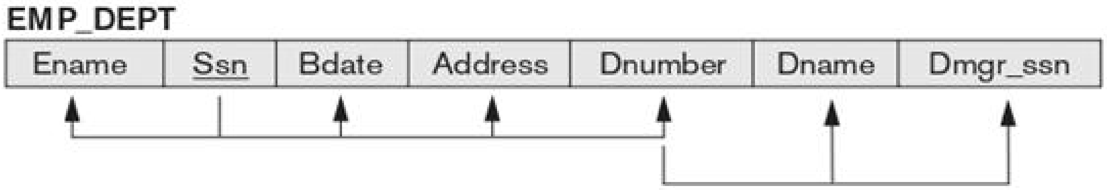
Figure: A relation state of TEACH with a possible functional dependency
TEXT → COURSE. However, TEACHER → COURSE,
TEXT → TEACHER and COURSE → TEXT are ruled out.

TEACH

TEACHER	COURSE	TEXT
Smith	Data Structures	Bartram
Smith	Database Systems	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Definition: The set of all dependencies that include ‘F’ as well as all dependencies that can be inferred from ‘F’ is called the closure of ‘F’, it is denoted by F^1.

For example, suppose that the se fo functional dependecies on the relational schema EMP_DEPT is



F = {Ssn → { Ename, Bdate, Address, Dnumber }
Dnumber → { Dname, Dmgr_ssn }}

Some additonal functional dependencies that we can infer from ‘F’ are the following:

Ssn → { Dname, Dmgr_ssn }
Ssn → Ssn
Dnumber → Dname

To determine a systematic way to infet dependencies we must discover a set of **inference rules** that can be used to infer new dependencies from a given set of dependencies.

The FD{X,Y} → Z is abbreviated to XY→Z and the FD{X,Y,Z}→ {U,V} is abbreviated to XYZ→UV.

The following 6 rules IR1 through IR6 are well known inference rules for functional dependencies.

IR1(Reflexive rule): If $X \supseteq Y$ then $X \rightarrow Y$

IR2(Augmentarion rile): $\{X \rightarrow Y\} = XZ \rightarrow YZ$

IR3(Transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} = X \rightarrow Z$

IR4(Decomposition rule): $\{X \rightarrow YZ\} = X \rightarrow Y$

IR5(Union rule): $\{X \rightarrow Y, X \rightarrow Z\} = X \rightarrow YZ$

IR6(Pseudotransitive rule): $\{ X \rightarrow Y, WY \rightarrow Z \} = \{WX \rightarrow Z\}$

Equivalence of Sets of Functional Dependencies:

A set of functinal dependencies 'F' is said to cover another set of functional dependencies 'E' if every FD in E is also F, i.e., if every dependency in E can be inferred from F alternatively we can say that E is covered by F.

Definition: Two sets of functional dependencies E and F are equivalent if E* = F*. Therefore, equivelence means that every FD in can be inferred from F and every FD in F can be inferred from E, that is E id equivalent to F if both conditions E covers F and F covers E hold.

Minimal Sets of Functional Dependencies:

A minimal cover of a set of functinal dependencies E is a set of functional dependencies F that satisfies the properly that every dependency in E is in the closure F* of F.

Definition: A minimal cover of a set of functional dependencies E is a minimal set of dependencies that is equivalent to E.

Normal Forms Based on Primary Keys

A set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the **normalization** process for relational schema design.

Normalization: In database design process the table is the basic building block. The ER modeling gives good table structures. But it is possible to create poor table.

Normalization is an analysis of functional dependency between the attributes of a relation; it reduces the complex user views into set of stable subgroups or fields.

The normalization process, as first proposed by Codd (1972a), This process is used to create good table structures to minimize data redundancies. The process, which proceeds in a top-down fashion.

Normalization works through a series of stages called normal forms. Initially, Codd proposed three normal forms, which he called first, second, and third normal form (1NF, 2NF, 3NF) . A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies. It can be considered as a "filtering" or "purification" process to make the design have successively better quality. An unsatisfactory relation schema that does not meet the condition for a normal form.

Definition: The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Practical Use of Normal Forms:

Most practical design projects in commercial and governmental environment acquire existing designs of databases from previous designs, from designs in legacy models, or from existing files. Existing designs are evaluated by applying the tests for normal forms, and normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and 5NF. Designers and users must either already know them or discover them as a part of the business. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers need not normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF.

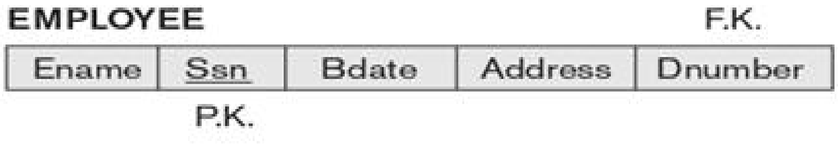
Definition: Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

Definitions of Keys and Attributes Participating in Keys:

The definitions of keys of a relation schema:

Definition: A **superkey** of a relation schema $R = \{A1, A2, \dots , An\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples $t1$ and $t2$ in any legal relation state r of R will have $t1[S] = t2[S]$. A key K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey anymore.

The difference between a key and a superkey is that a key has to be minimal; that is, if we have a key $K = \{A1, A2, \dots , Ak\}$ of R, then $K - \{Ai\}$ is not a key of R for any Ai, $1 \le i \le k$. In following Figure, {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.



If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called secondary keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In the above Figure, {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key.

Definition: An attribute of relation schema R is called a **prime attribute** of R if it is a member of some candidate key of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

First Normal Form:

1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. The only attribute values permitted by 1NF are single atomic (or indivisible) values.

Definition: A relation is said to be in first normal form it is already in unnormalized form and it has ***no repeating groups***.

Consider the DEPARTMENT relation whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in the following Figure (a). We assume that each department can have a number of locations. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure (b).

Figure: a) A relation schema that is not in 1NF.

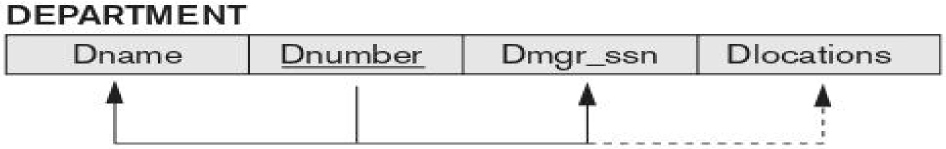
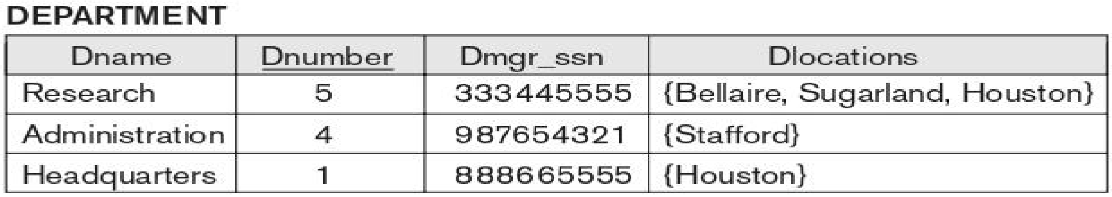


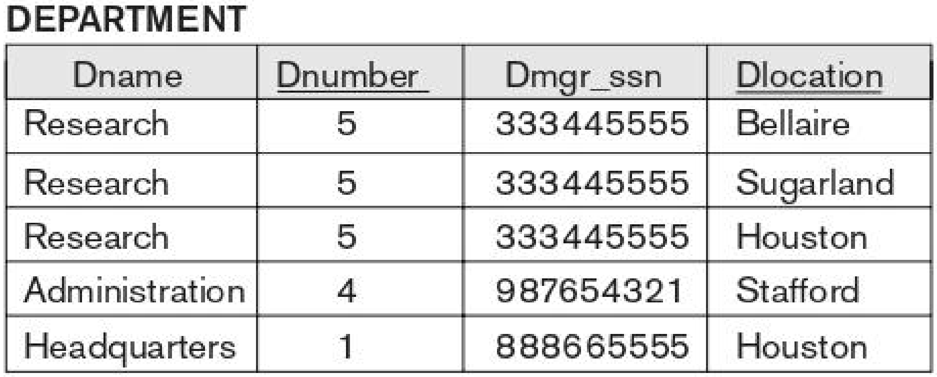
Figure:b) Sample state of relation DEPARTMENT.


DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

The DEPARTMENT relation to achieve 1NF, expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure (c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing redundancy in the relation and hence is rarely adopted.

Figure: c) 1NF version of the same relation with redundancy.



DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Second Normal Form:

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency X → Y is a full functional dependency .

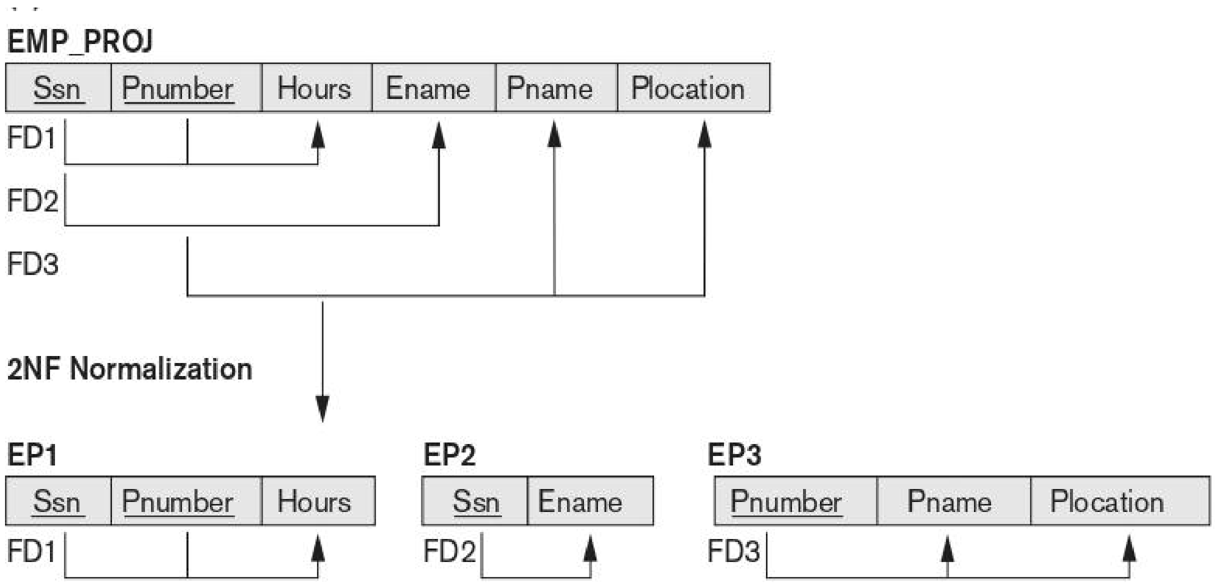
Definition: A relation schema R is in 2NF if every nonprime attribute A in R is ***fully functionally dependent*** on the primary key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all.

The EMP_PROJ relation is in 1NF but is not in 2NF. The nonprime attribute **Ename** violates 2NF because of FD2, as do the nonprime attributes **Pname** and **Plocation** because of FD3. Each of the functional dependencies FD2 and FD3 violates 2NF because **Ename** can be functionally determined by only **Ssn**, and both **Pname** and **Plocation** can be functionally determined by only **Pnumbe**r. Attributes **Ssn** and **Pnumber** are a part of the primary key {**Ssn, Pnumber**} of EMP_PROJ, thus violating the 2NF test.

Therefore, the functional dependencies FD1, FD2, and FD3 in the follwing Figure lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 each of which is in 2NF.

Figure: Normalizing into 2NF. Normalizing EMP_PROJ into 2NF relations.



Third Normal Form:

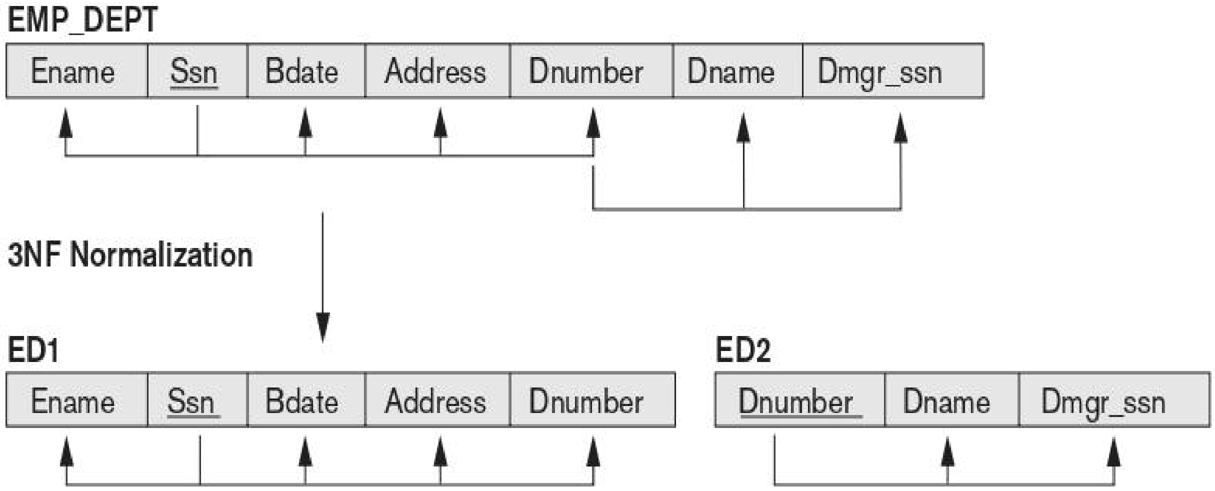
Third normal form (3NF) is based on the concept of ***transitive dependency***. A functional dependency X → Y in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold.

Definition: According to Codd's original definition, a relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is ***transitively dependent*** on the primary key.

The relation schema EMP_DEPT is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of **Dmgr_ssn (and also Dname**) on **Ssn** via **Dnumber**. The dependency **Ssn** → **Dmgr_ssn** is transitive through **Dnumber** in EMP_DEPT because both the dependencies **Ssn** → **Dnumber** and **Dnumber** → **Dmgr_ssn** hold.

We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in the following Figure. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Figure: Normalizing into 3NF. Normalizing EMP_DEPT into 3NF relations.



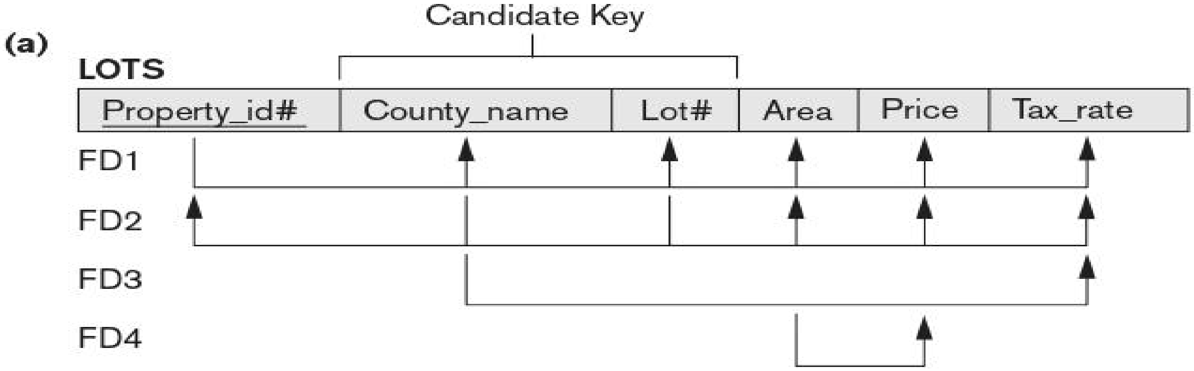
Definitions of Second and Third Normal Forms

General Definition of Second Normal Form:

Definition. A relation schema R is in second normal form (2NF) if every nonprime attribute A in R is not partially dependent on any key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure (a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}.

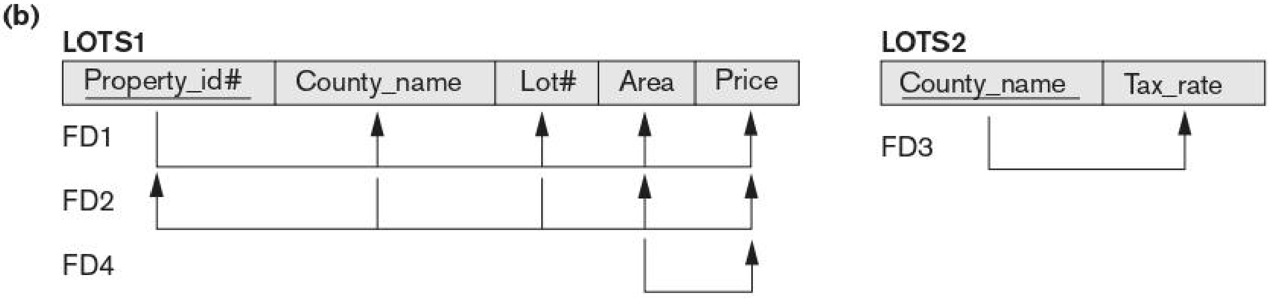
Figure (a): The LOTS relation with its functional dependencies FD1 through FD4



Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 in Figure (a) hold. We choose Property_id# as the primary key, so it is underlined in Figure (a). Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate
FD4: Area → Price

Figure (b): Decomposing into the 2NF relations LOTS1 and LOTS2.



The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure (b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

General Definition of Third Normal Form:

Definition. A relation schema R is in third normal form (3NF) if, whenever a nontrivial functional dependency X → A holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

According to this definition, LOTS2 (Figure (b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure (c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Figure (c): Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B.

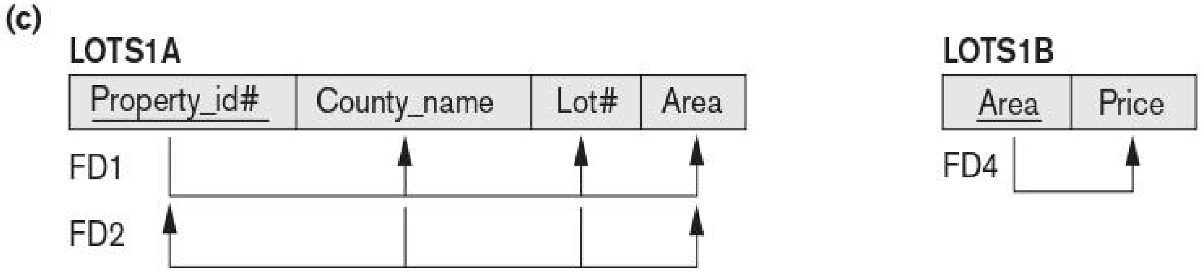
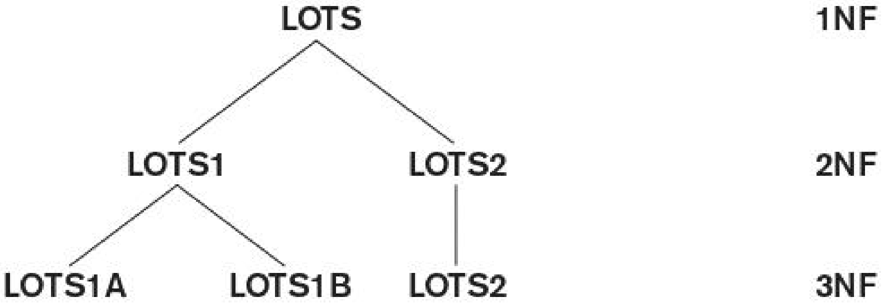


Figure: Progressive normalization of LOTS into a 3NF design.



Alternative Definition: A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

- ■ It is fully functionally dependent on every key of R.
- ■ It is nontransitively dependent on every key of R.

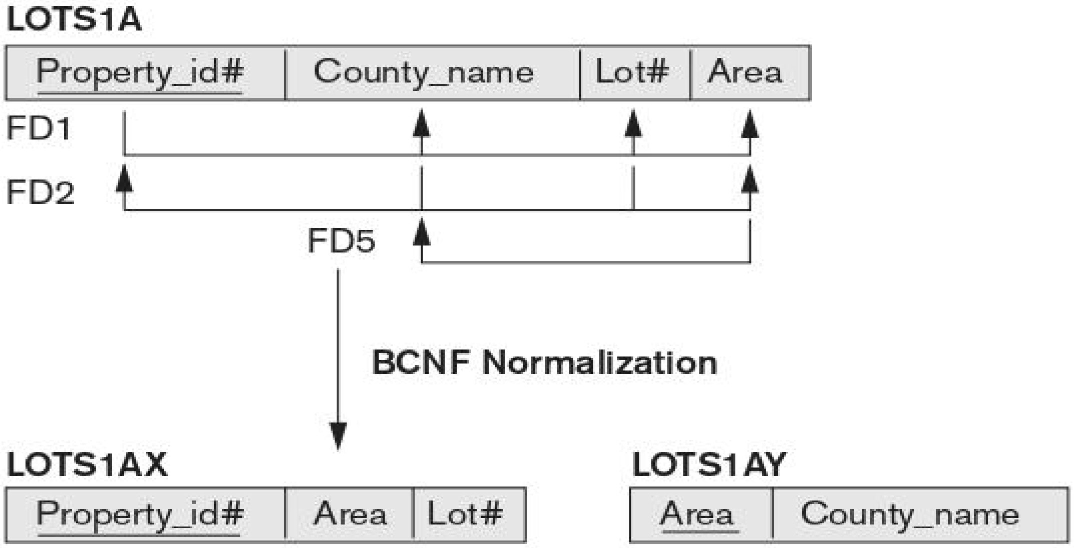
Boyce-Codd Normal Form (BCNF)

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF.

BCNF is the advanced version of 3NF. A table is in BCNF if every functional dependency X->Y, X is the super key of the table. For BCNF, the table should be in 3NF, and for every FD. LHS is super key.

Definition: A relation schema R is in **BCNF** if whenever a ***nontrivial*** functional dependency X → A holds in R, then X is a superkey of R.

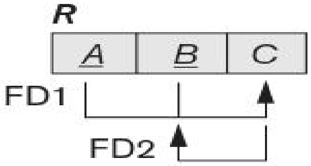
Figure: BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition.



In our example, FD5 violates BCNF in LOTS1A because Area is not a superkey of LOTS1A. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

Most relation schemas that are in 3NF are also in BCNF. Only if there exists some f.d. X → A that holds in a relation schema R with X not being a superkey and A being a prime attribute will R be in 3NF but not in BCNF. The relation schema R shown in the following Figure illustrates the general case of such a relation. Such an f.d. leads to potential redundancy of data, as we illustrated above in case of FD5: Area → County_name.in LOTS1A relation.

Figure: A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d.C → B.



Properties of Relational Decompositions

The process of decomposition that we used to get rid of unwanted dependencies and achieve higher normal forms. A set of relations that together form the relational database schema must possess certain additional properties to ensure a good design.

1. **Relation Decomposition and Insufficiency of Normal Forms:**

A single universal relation schema R = {A1, A2, … , An} that includes all the attributes of the database. We implicitly make the universal relation assumption, which states that every attribute name is unique. The set F of functional dependencies that should hold on the attributes of R is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas D = {R1, R2, … , Rm} that will become the relational database schema; D is called a decomposition of R.

We must make sure that each attribute in R will appear in at least one relation schema Ri in the decomposition so that no attributes are lost; formally, we have

$$\bigcup_{i=1}^{m} R_i = R$$

This is called the attribute preservation condition of a decomposition.

2. Dependency Preservation Property of a Decomposition:

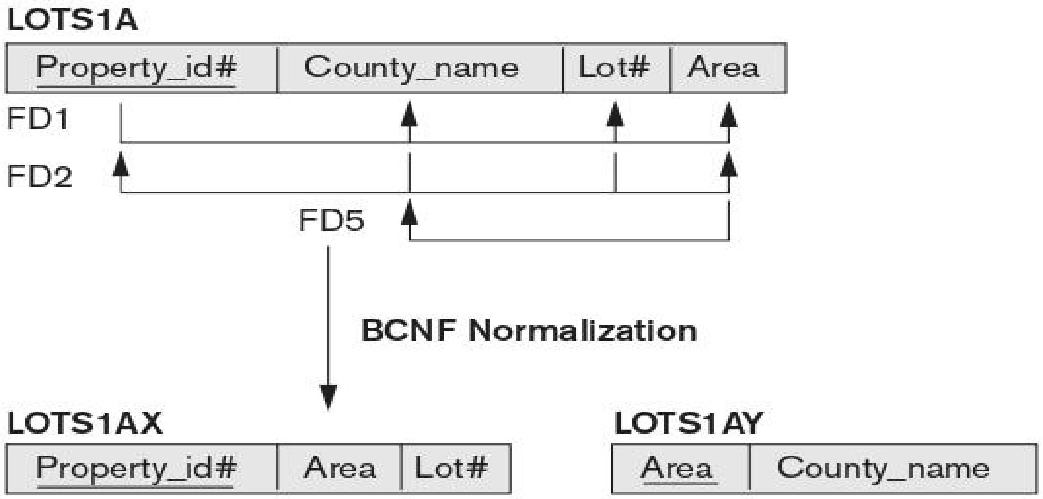
The functional dependency X → Y specified in F either appeared directly in one of the relation schemas Ri in the decomposition D or could be inferred from the dependencies that appear in some Ri. Informally, this is the ***dependency preservation condition.***

Definition: Given a set of dependencies F on R, the projection of F on Ri, denoted by π_{Ri} (F) where Ri is a subset of R, is the set of dependencies X → Y in F^+ such that the attributes in X ∪ Y are all contained in Ri. Hence, the projection of F on each relation schema Ri in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all the left- and right-hand-side attributes of those dependencies are in Ri. We say that a decomposition D = {R1, R2, … , Rm} of R is **dependency-preserving** with respect to F if the union of the projections of F on each Ri in D is equivalent to F; that is,

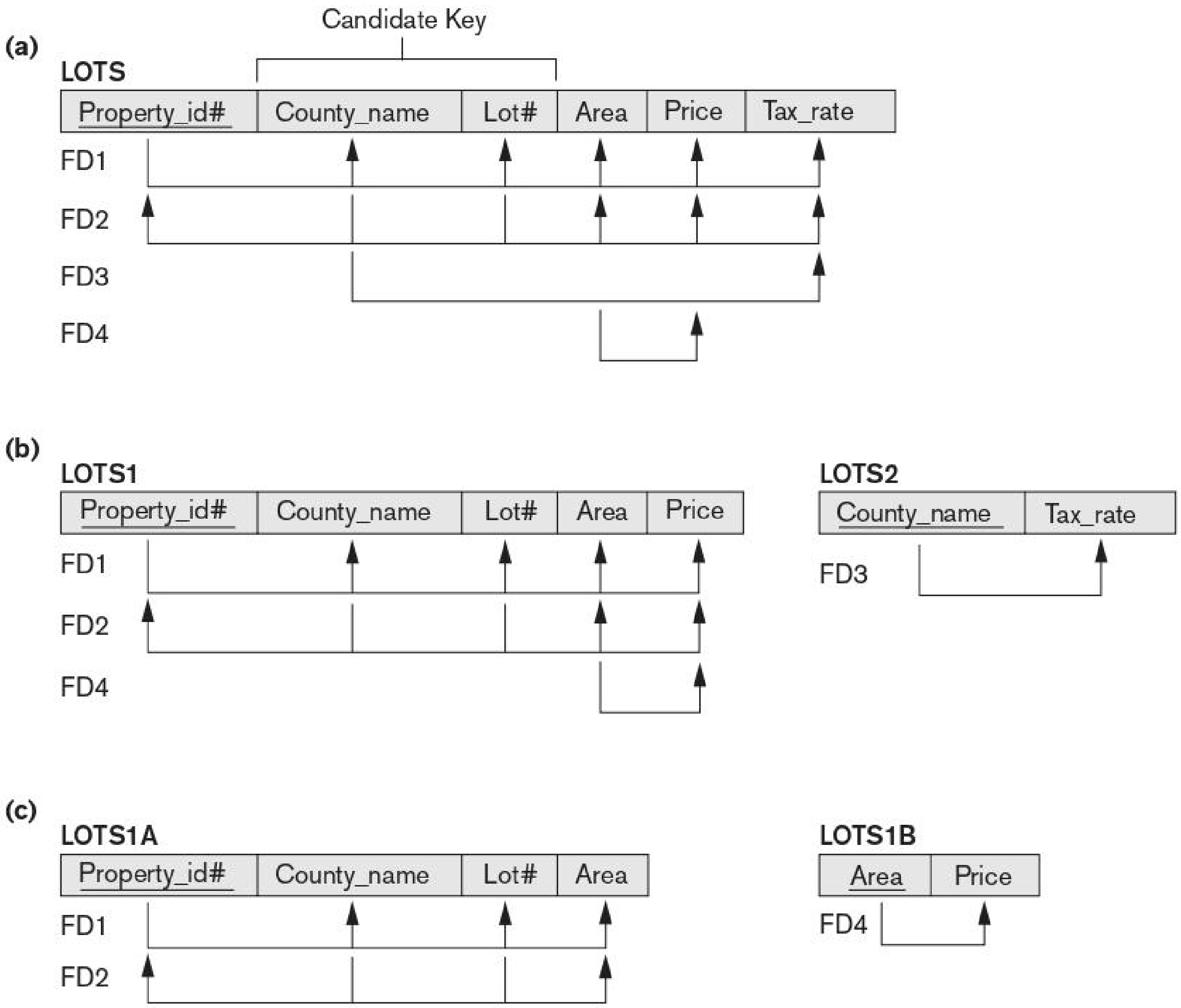
$$((\pi_{R1}(F)) \cup K \cup (\pi_{Rm}(F)))^+ = F^+.$$

If a decomposition is not dependency-preserving, some dependency is lost in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations.

An example of a decomposition that does not preserve dependencies is shown in Figure in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}.



The decompositions in the following Figure are dependency-preserving.



Claim 1. It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation Ri in D is in 3NF.

3. Nonadditive (Lossless) Join Property of a Decomposition:

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition.

Definition: Formally, a decomposition D = {R1, R2, … , Rm} of R has the lossless (nonadditive) join property with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F, the following holds, where * is the NATURAL JOIN of all the relations in D: $*(\pi_{R1}(r), \dots, \pi_{Rm}(r)) = r$.

The word loss in lossless refers to loss of information, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous or invalid information.

4. Testing Binary Decompositions for the Nonadditive Join Property:

A particular decomposition D into **n** relations obeys the nonadditive join property with respect to a set of functional dependencies F. There is a special case of a decomposition called a **binary decomposition**—decomposition of a relation R into two relations. A test called the NJB (Nonadditive Join Test for Binary Decompositions) property test was used to do binary decomposition of the relation, which met 3NF but did not meet BCNF, into two relations that satisfied this property.

5. Successive Nonadditive Join Decompositions:

The successive decomposition of relations during the process of second and third normalization. To verify that these decompositions are nonadditive, we need to ensure another property, as set forth in Claim 2.

Claim 2 (Preservation of Nonadditivity in Successive Decompositions): If a decomposition D = {R_1, R_2, … , R_m} of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition Di = {Q_1, Q_2, … , Q_k} of Ri has the nonadditive join property with respect to the projection of F on R_i, then the decomposition D2 = {R_1, R_2, … , R_{i-1}, Q_1, Q_2, … , Q_k, R_{i+1}, … , R_m} of R has the nonadditive join property with respect to F.

Algorithms for Relational Database Schema Design

Two algorithms for creating a relational decomposition from a universal relation. The first algorithm decomposes a universal relation into dependencypreserving 3NF relations that also possess the nonadditive join property. The second algorithm decomposes a universal relation schema into BCNF schemas that possess the nonadditive join property.

1. Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

By now we know that it is not possible to have all three of the following: (1) guaranteed nonlossy (nonadditive) design, (2) guaranteed dependency preservation, and (3) all relations in BCNF.
Now we give an algorithm where we achieve conditions 1 and 2 and only guarantee 3NF. Algorithm 1 yields a decomposition D of R that does the following:

- Preserves dependencies
- Has the nonadditive join property
- Is such that each resulting relation schema in the decomposition is in 3NF

Algorithm 1: Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F.
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\} \}$, where $X \rightarrow A_1$, $X \rightarrow A_2$, … , $X \rightarrow A_k$ are the only dependencies in G with X as left-hand side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R.
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S.

Example of Algorithm 1: Consider the following universal relation:

U (Emp_ssn, Pno, Esal, Ephone, Dno, Pname, Plocation)

Emp_ssn, Esal, and Ephone refer to the Social Security number, salary, and phone number of the employee. Pno, Pname, and Plocation refer to the number, name, and location of the project. Dno is the department number.

The following dependencies are present:

FD1: Emp_ssn → {Esal, Ephone, Dno}
FD2: Pno → { Pname, Plocation}
FD3: Emp_ssn, Pno → {Esal, Ephone, Dno, Pname, Plocation}

By virtue of FD3, the attribute set {Emp_ssn, Pno} represents a key of the universal relation. Hence F, the set of given FDs, includes {Emp_ssn → Esal, Ephone, Dno; Pno → Pname, Plocation; Emp_ssn, Pno → Esal, Ephone, Dno, Pname, Plocation}.

By applying the minimal cover Algorithm, in step 3 we see that Pno is an extraneous attribute in Emp_ssn, Pno → Esal, Ephone, Dno. Moreover, Emp_ssn is extraneous in Emp_ssn, Pno → Pname, Plocation. Hence the minimal cover consists of FD1 and FD2 only asfollows:

Minimal cover G: {Emp_ssn → Esal, Ephone, Dno; Pno → Pname, Plocation}

The second step of Algorithm produces relations R1 and R2 as:

R1 (Emp_ssn, Esal, Ephone, Dno)
R2 (Pno, Pname, Plocation)

In step 3, we generate a relation corresponding to the key {Emp_ssn, Pno} of U. Hence, the resulting design contains:

R1 (Emp_ssn, Esal, Ephone, Dno)
R2 (Pno, Pname, Plocation)
R3 (Emp_ssn, Pno)

This design achieves both the desirable properties of dependency preservation and nonadditive join.

2. Nonadditive Join Decomposition into BCNF Schemas

The next algorithm decomposes a universal relation schema R = {A_1, A_2, … , A_n} into a decomposition D = {R_1, R_2, … , R_m} such that each R_i is in BCNF and the decomposition D has the lossless join property with respect to F. Algorithm 2 utilizes property NJB and claim 2 (preservation of nonadditivity in successive decompositions) to create a nonadditive join decomposition D = {R_1, R_2, … , R_m} of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in BCNF.

Algorithm 2: Relational Decomposition into BCNF with Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

```
1. Set D := {R} ;
2. While there is a relation schema Q in D that is not in BCNF do
     {
       choose a relation schema Q in D that is not in BCNF;
       find a functional dependency X → Y in Q that violates BCNF;
       replace Q in D by two relation schemas (Q − Y) and (X ∪ Y);
     } ;
```

Each time through the loop in Algorithm 2, we decompose one relation schema Q that is not in BCNF into two relation schemas. According to property NJB for binary decompositions and claim 2, the decomposition D has the nonadditive join property. At the end of the algorithm, all relation schemas in D will be in BCNF. We illustrated the application of this algorithm to the TEACH relation schema is decomposed into TEACH1(Instructor, Student) and TEACH2(Instructor, Course) because the dependency FD2 Instructor → Course violates BCNF.

Indexing Structures for Files

Types of Single-Level Ordered Indexes

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field** (or **indexing attribute**). The values in the index are ordered so that we can do a binary search on the index.

There are several types of ordered indexes. A **primary index** is specified on the ordering key field of an ordered file of records. An ordering key field is used to physically order the file records on disk, and every record has a unique value for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field— another type of index, called a **clustering index**, can be used. The data file is called a **clustered file** in this latter case. A third type of index, called a **secondary index**, can be specified on any nonordering field of a file.

1. Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file. The first field is of the same data type as the ordering key field called the **primary key** of the data file, and the second field is a pointer to a disk block (a block address). There is one **index entry** (or index record) in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values.

Figure 17.1 illustrates this primary index. The total number of entries in the index is the same as the number of disk blocks in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.

Indexes can also be characterized as dense or sparse. A **dense** index has an index entry for every search key value (and hence every record) in the data file. A **sparse** (or **nondense**) index, on the other hand, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file.

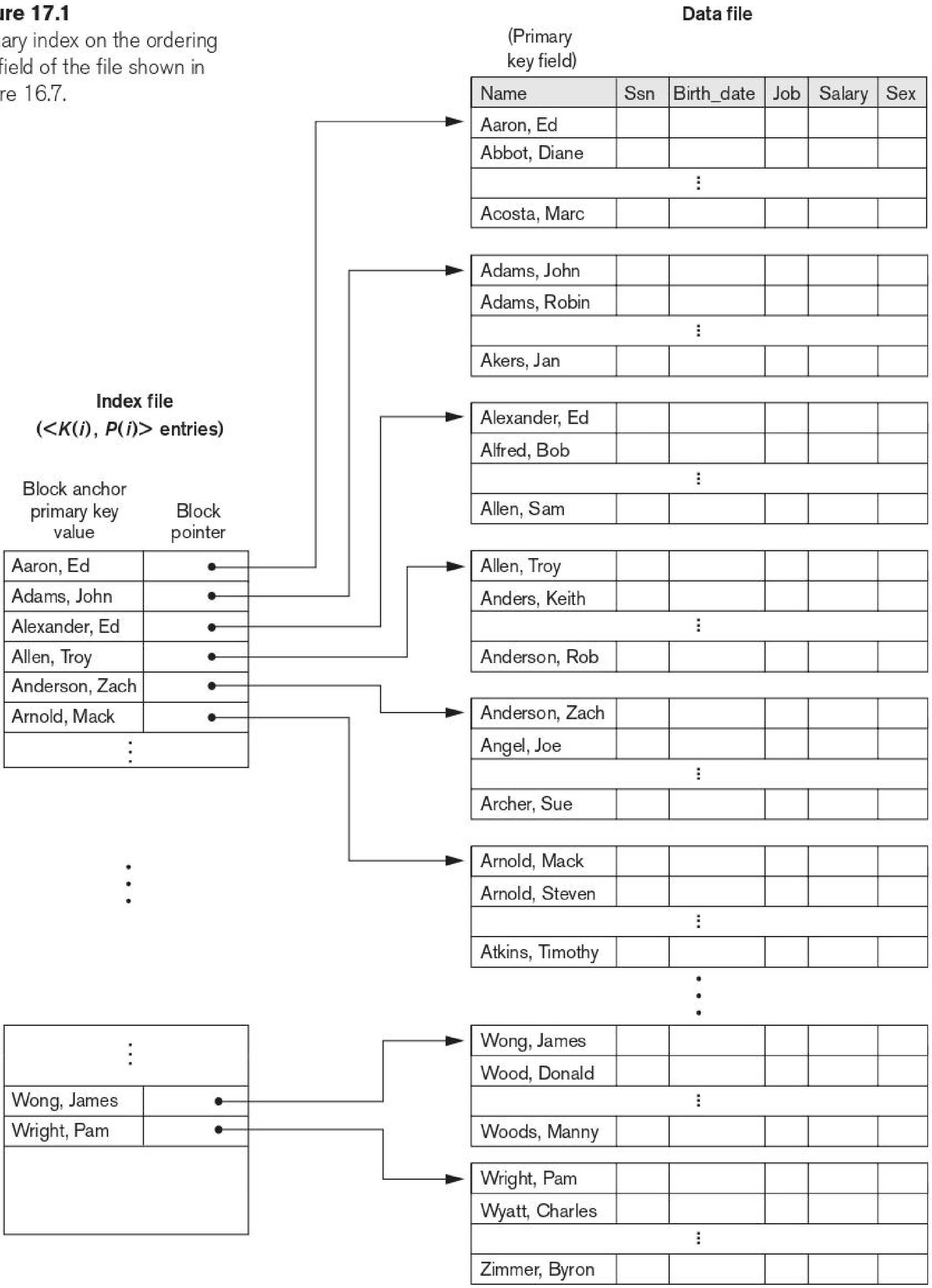


Figure 17.1
Primary index on the ordering key field of the file shown in Figure 16.7.

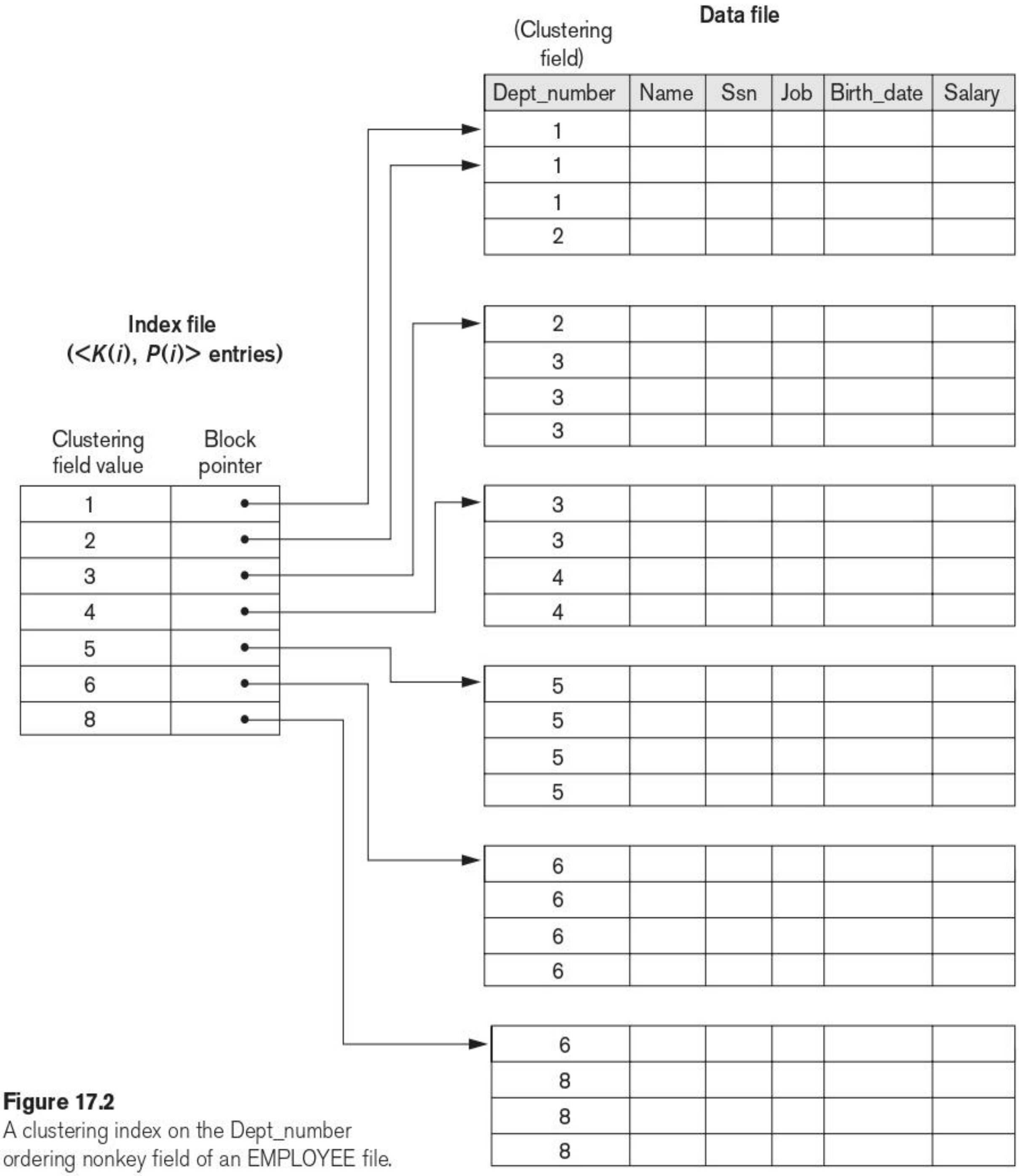
A major problem with a primary index—as with any ordered file—is insertion and deletion of records. With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since moving records will change the anchor records of some blocks.

2. Clustering Indexes

If file records are physically ordered on a nonkey field—which does not have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file**. We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a distinct value for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each distinct value of the clustering field, and it contains the value and a pointer to the first block in the data file that has a record with that value for its clustering field. Figure 17.2 shows an example.

Notice that record insertion and deletion still cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block for each value of the clustering field; all records with that value are placed in the block. This makes insertion and deletion relatively straightforward. Figure 17.3 shows this scheme.



Clustering field value	Block pointer
1	●
2	●
3	●
4	●
5	●
6	●
8	●

Dept_number	Name	Ssn	Job	Birth_date	Salary
1					
1					
1					
2					

2					
3					
3					
3					

3					
3					
4					
4					

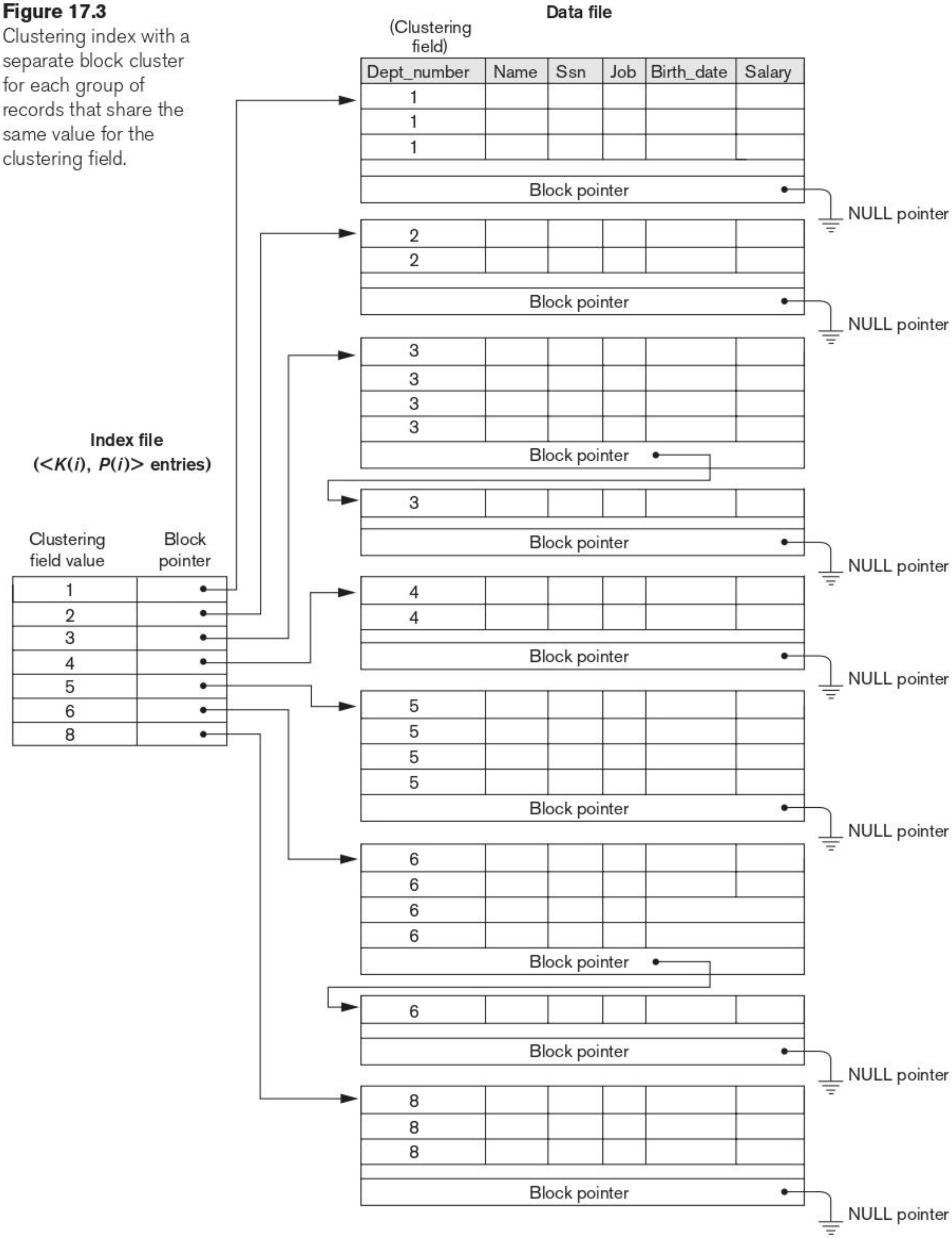
5					
5					
5					
5					

6					
6					
6					
6					

6					
8					
8					
8					

Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Figure 17.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



3. Secondary Indexes

A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values. The index is again an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record pointer*. Many secondary indexes can be created for the same file.

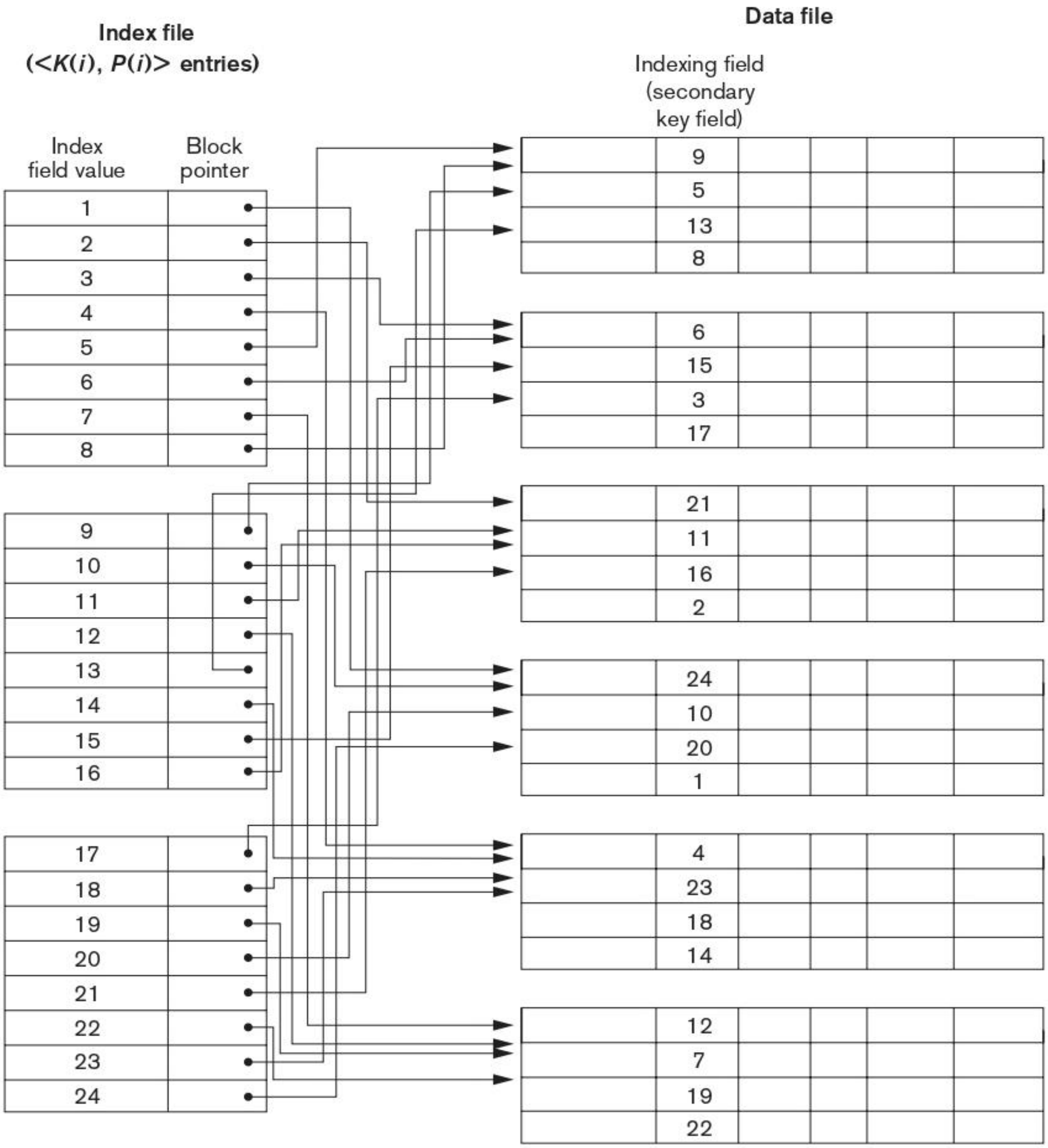
First we consider a secondary index access structure on a key (unique) field that has a distinct value for every record. Such a field is sometimes called a **secondary key**. A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries.

Figure 17.4 illustrates a secondary index in which the pointers P(i) in the index entries are block pointers, not record pointers. Once the appropriate disk block is transferred to a main memory buffer, a search for the desired record within the block can be carried out.

Notice that a secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field. The primary and clustering indexes assume that the field used for **physical ordering** of records in the file is the same as the indexing field.

If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 17.5.

Figure 17.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



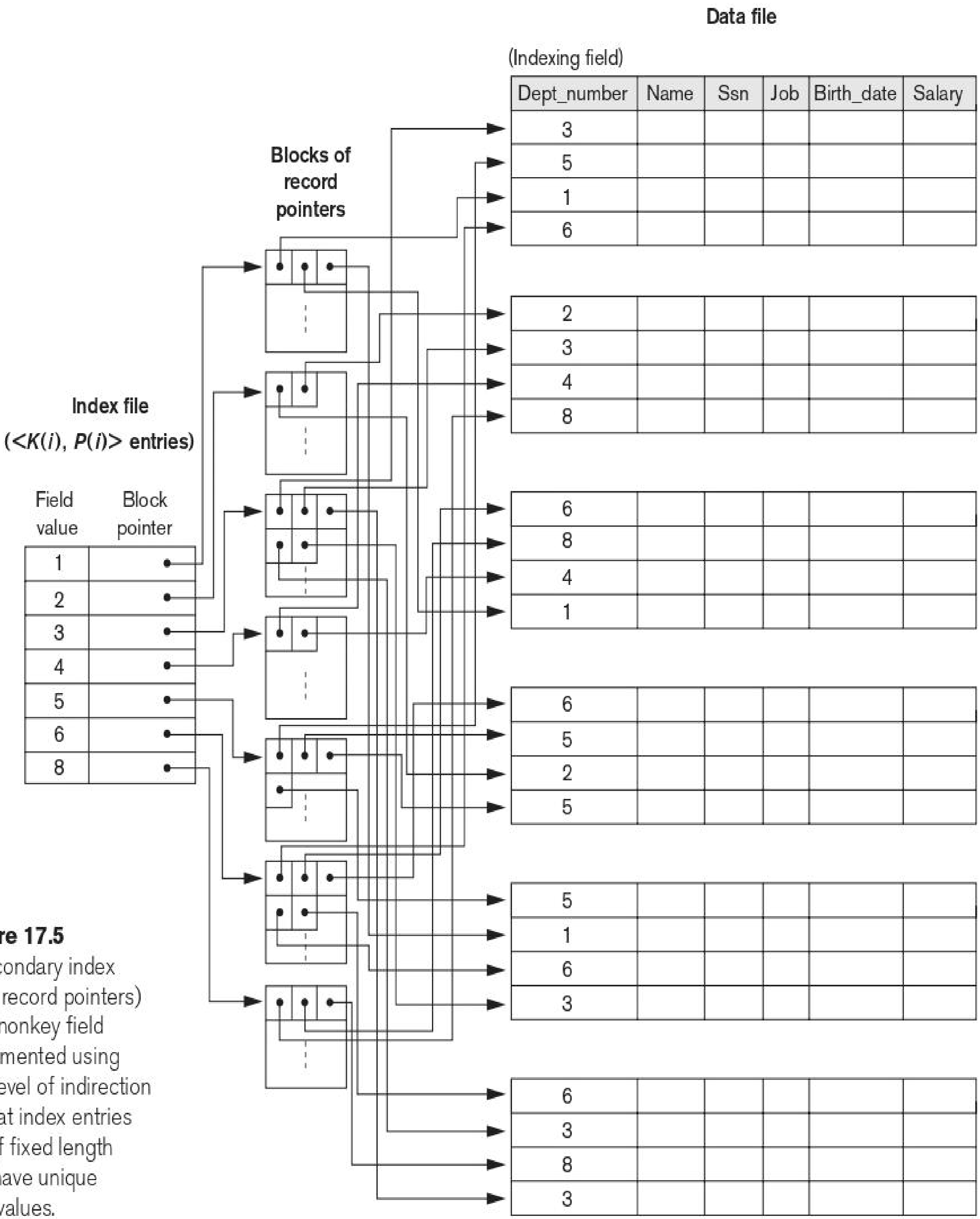


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A **multilevel index** considers the index file, which we will now refer to as the first (or base) level of a multilevel index, as an ordered file with a distinct value for each K(i). Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first level. The blocking factor **bfri** for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address. If the first level has r1 entries, and the blocking factor—which is also the **fan-out**—for the index is **bfri = fo**, then the first level needs [(r1/fo)] blocks, which is therefore the number of entries r2 needed at the second level of the index.

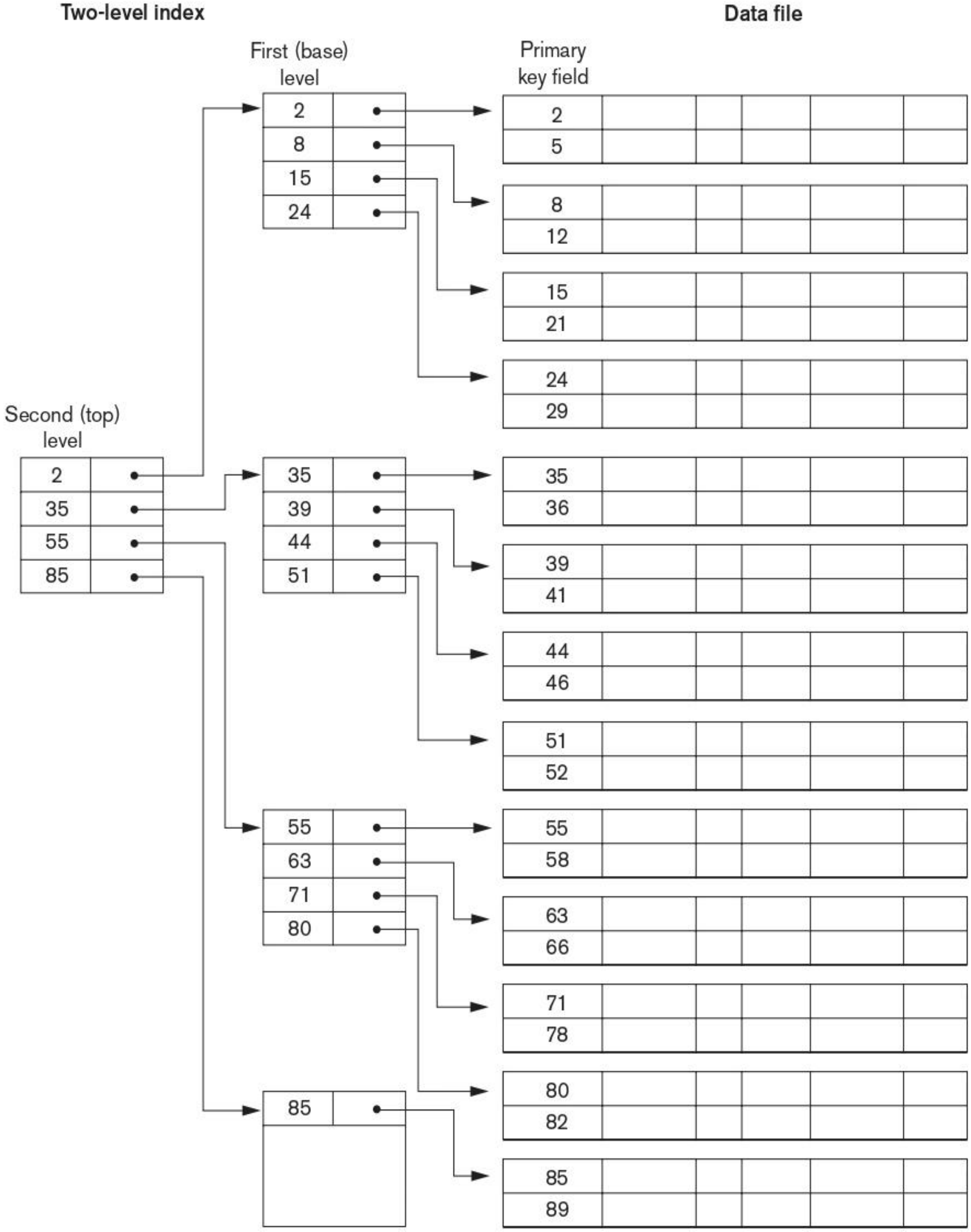
We can repeat this process for the second level. The third level, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is r3 = [(r2/fo)].

Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the tth level is called the top index level.

Hence, a multilevel index with r1 first-level entries will have approximately **t** levels, where **t** = [(logfo(r1))]. When searching the index, a single disk block is retrieved at each level. Hence, **t** disk blocks are accessed for an index search, where **t** is the number of index levels.

The multilevel scheme described here can be used on any type of index—whether it is primary, clustering, or secondary—as long as the first-level index has distinct values for K(i) and fixed-length entries. Figure 17.6 shows a multilevel index built over a primary index.

Figure 17.6
A two-level primary index resembling ISAM (indexed sequential access method) organization.



A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an **indexed sequential file** and was used in a large number of early IBM systems.

Algorithm:. Searching a Nondense Multilevel Primary Index with t Levels

```
(*We assume the index entry to be a block anchor that is the first key per block*)
 p ← address of top-level block of index;
 for j ← t step − 1 to 1 do
      begin
           read the index block (at jth index level) whose address is p;
           search block p for entry i such that Kj (i) ≤ K < Kj (i + 1)
   (* if Kj (i)
           is the last entry in the block, it is sufficient to satisfy Kj (i) ≤ K *);
           p ← Pj (i ) (* picks appropriate pointer at jth index level *)
   end;
   read the data file block whose address is p;
   search block p for record with key = K;
```

A multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value. We are still faced with the problems of dealing with index insertions and deletions, because all index levels are physically ordered files. To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks.

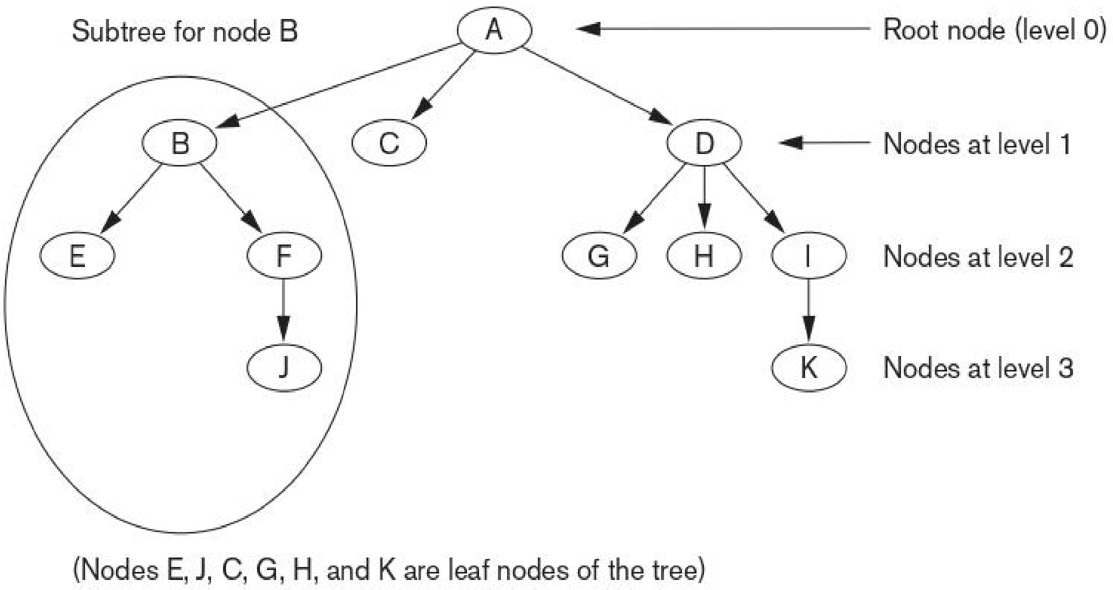
Dynamic Multilevel Indexes Using B-Trees and B+-Trees

B-trees and B+-trees are special cases of the well-known search data structure known as a **tree**. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being zero.

A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node n and the subtrees of all the child nodes of **n**. Figure illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes. Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**.

Figure:

A tree data structure that shows an unbalanced tree.



(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

1. Search Trees and B-Trees:

A search tree is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields.

Search Trees:

A search tree is slightly different from a multilevel index. A **search tree of order** p is a tree such that each node contains at most p − 1 search values and p pointers in the order <P1,K1,P2,K2,…,Pq-1,Kq-1,Pq>, where q ≤ p. Each Pi is a pointer to a child node (or a NULL pointer), and each Ki is a search value from some ordered set of values. All search values are assumed to be unique. **Figure 1** illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, K1 < K2 < … < Kq−1.
2. For all values X in the subtree pointed at by Pi, we have Ki−1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki−1 < X for i = q (see Figure 1).

Whenever we search for a value X, we follow the appropriate pointer Pi according to the formulas in condition 2 above. **Figure 2** illustrates a search tree of order p = 3 and integer search values. Notice that some of the pointers Pi in a node may be NULL pointers.

Figure 1: A node in a search tree with pointers to subtrees below it.

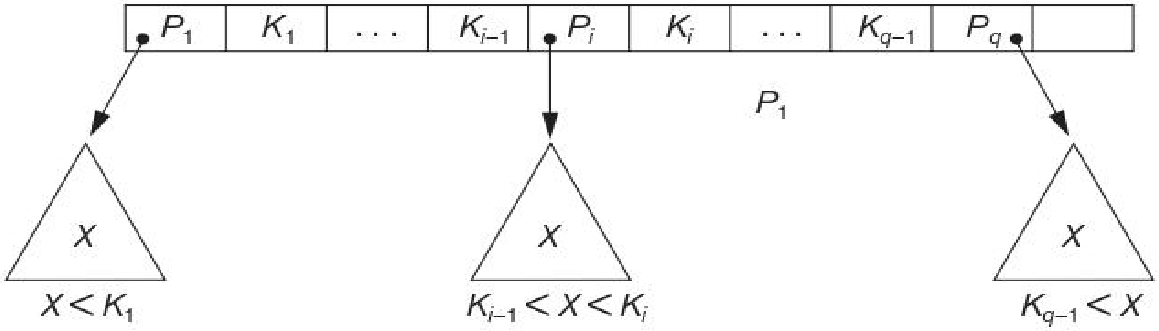
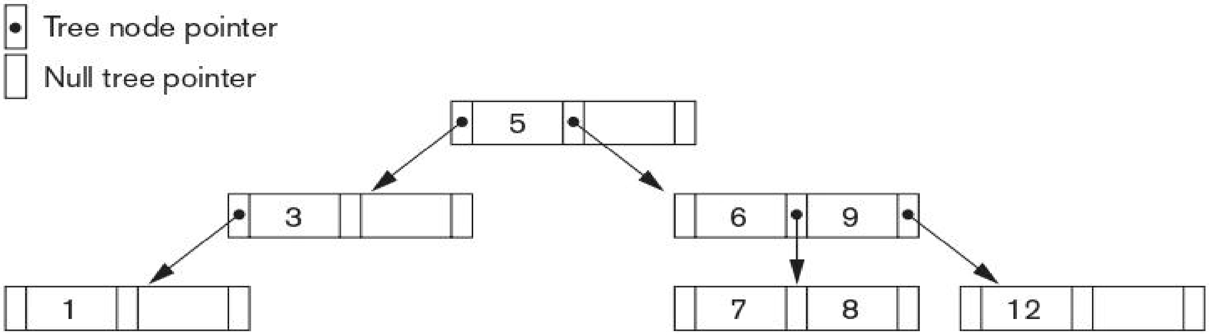


Figure 2: A search tree of order p = 3.



B-Trees.

The **B-tree** has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. More formally, a B-tree of order p, when used as an access structure on a key field to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure (a)) is of the form
   <P1, <K1,Pr1>, P2, <K2,Pr2>,… , <Kq-1,Prq-1> , Pq>

   where $q \leq p$. Each Pi is a **tree pointer**—a pointer to another node in the B-tree. Each Pri is a **data pointer**—a pointer to the record whose search key field value is equal to Ki (or to the data file block containing that record).

2. Within each node, $K1 < K2 < \ldots < Kq-1$.

3. For all search key field values X in the subtree pointed at by Pi (the ith subtree, see Figure 17.10(a)), we have:

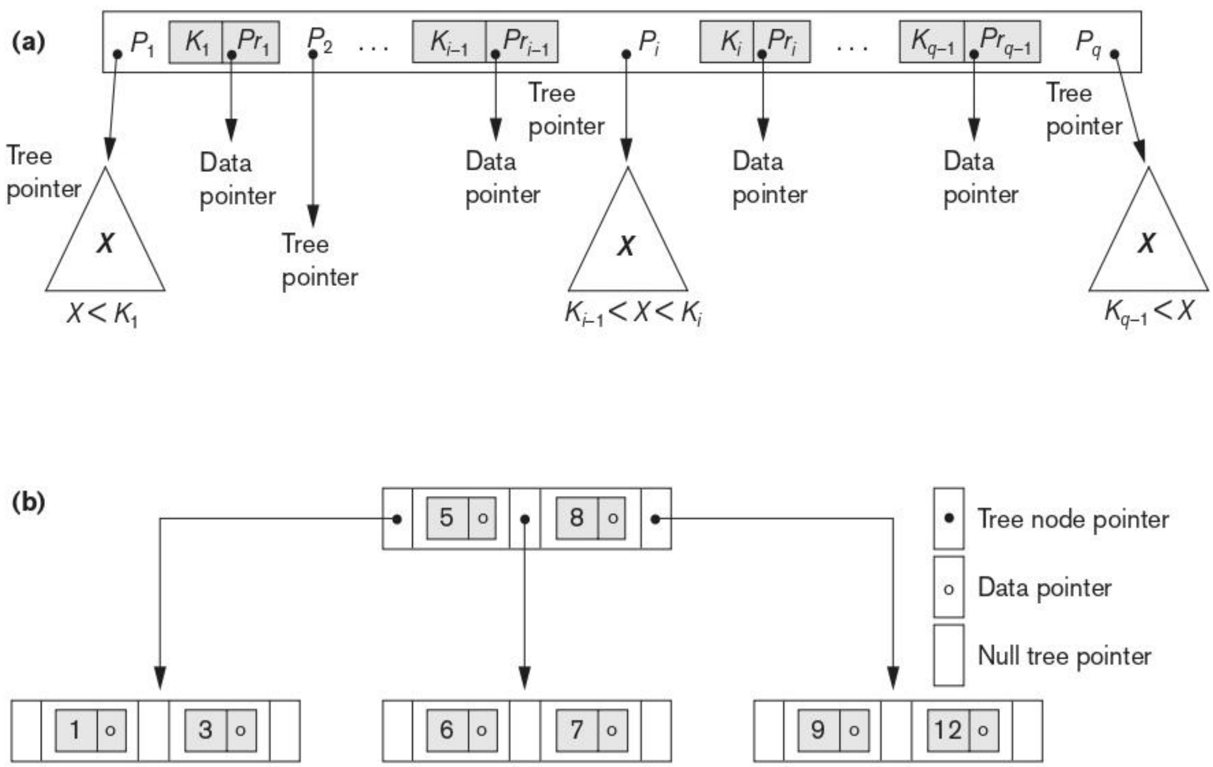
   $Ki-1 < X < Ki$ for $1 < i < q$; $X < Ki$ for $i = 1$; and $Ki-1 < X$ for $i = q$
4. Each node has at most p tree pointers.

5. Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).

7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers Pi are NULL.

Figure (b) illustrates a B-tree of order $p = 3$. Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field.

Figure: B-tree structures. (a) A node in a B-tree with q − 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.



2. B+-Trees:

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B+-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B+-tree, data pointers are stored only at the leaf nodes of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes.

The leaf nodes of the B+-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B+-tree correspond to the other levels of a multilevel index.

The structure of the internal nodes of a B+-tree of order p (Figure (a)) is as follows:

1. Each internal node is of the form

   <P1, K1, P2, K2,…, Pq-1, Kq-1, Pq>

   where q ≤ p and each Pi is a **tree pointer.**

2. Within each internal node, K1 < K2 < … < Kq−1.

3. For all search field values X in the subtree pointed at by Pi, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.

4. Each internal node has at most p tree pointers.

5. Each internal node, except the root, has at least [(p/2)] tree pointers. The root node has at least two tree pointers if it is an internal node.

6. An internal node with q pointers, q ≤ p, has q − 1 search field values.

The structure of the leaf nodes of a B+-tree of order p (Figure (b)) is as follows:

1. Each leaf node is of the form

   $<< K_1,Pr_1>, <K_2,Pr_2>,, \dots ,< K_{q-1},Pr_{q-1}> , P_{next}>$

   where q ≤ p, each Pri is a data pointer, and Pnext points to the next leaf node of the B+-tree.

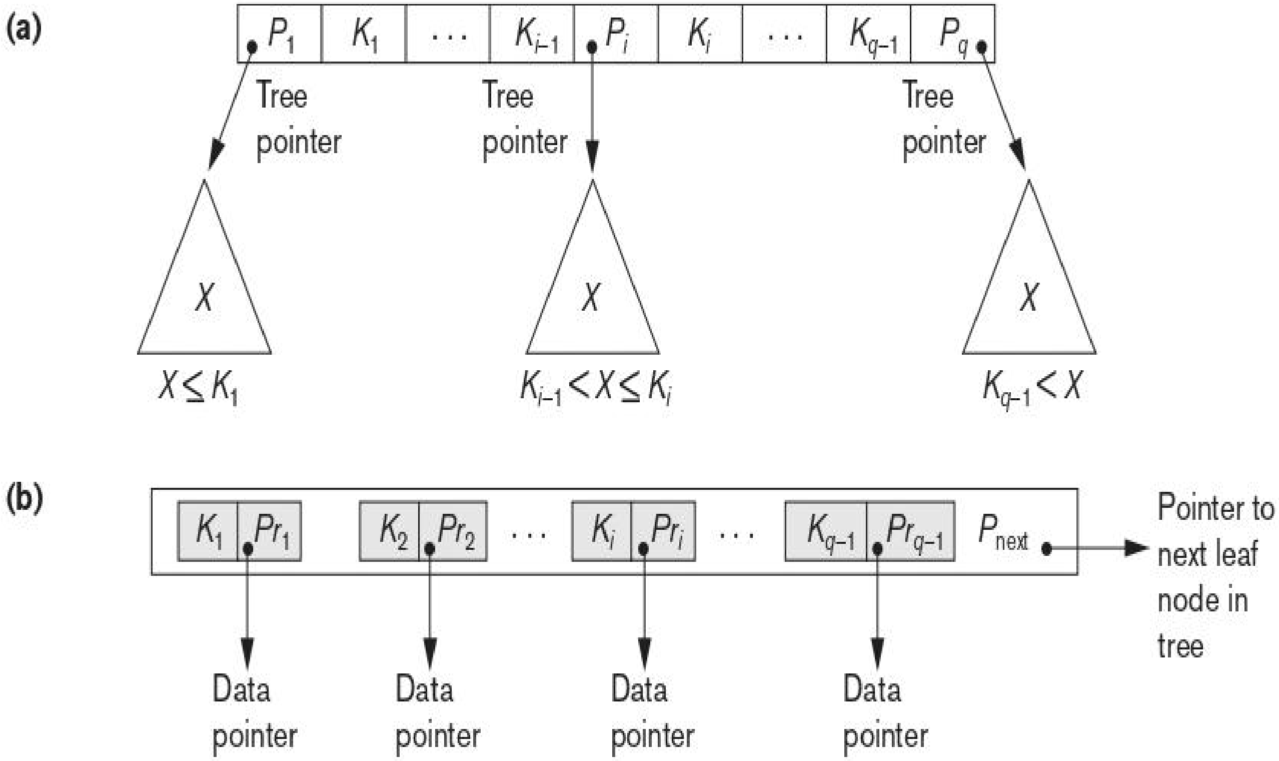
2. Within each leaf node, K1 ≤ K2 … , Kq−1, q ≤ p.

3. Each Pri is a **data pointer** that points to the record whose search field value is Ki or to a file block containing the record.

4. Each leaf node has at least [(p/2)] values.

5. All leaf nodes are at the same level.

Figure: The nodes of a B+-tree. (a) Internal node of a B+-tree with q − 1 search values. (b) Leaf node of a B+-tree with q − 1 search values and q − 1 data pointers.



Search, Insertion, and Deletion with B+-Trees:

The following Algorithm outlines the procedure using the B+-tree as the access structure to search for a record.

Algorithm: Searching for a Record with Search Key Field Value K, Using a B+- Tree

```
n ← block containing root node of B+-tree;
read block n;
while (n is not a leaf node of the B+-tree) do
    begin
     q ← number of tree pointers in node n;
     if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
         then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
         else if K > n.Kq−1
              then n ← n.Pq


                     else begin
                        search node n for an entry i such that n.Ki−1 < K ≤n.Ki ;
                          n ← n.Pi
                          end;
    read block n
    end;
search block n for entry (Ki , Pri ) with K = Ki ; (* search leaf node *)
if found
      then read data file block with address Pri and retrieve record
      else the record with search field value K is not in the data file;
```

Inserting a record in a file with a B+-tree existence of a key search field, and they must be modified appropriately for the case of a B+-tree on a nonkey field. We illustrate insertion and deletion with an example.

Fighre: An example of insertion in a B+-tree with p = 3 and pleaf = 2.

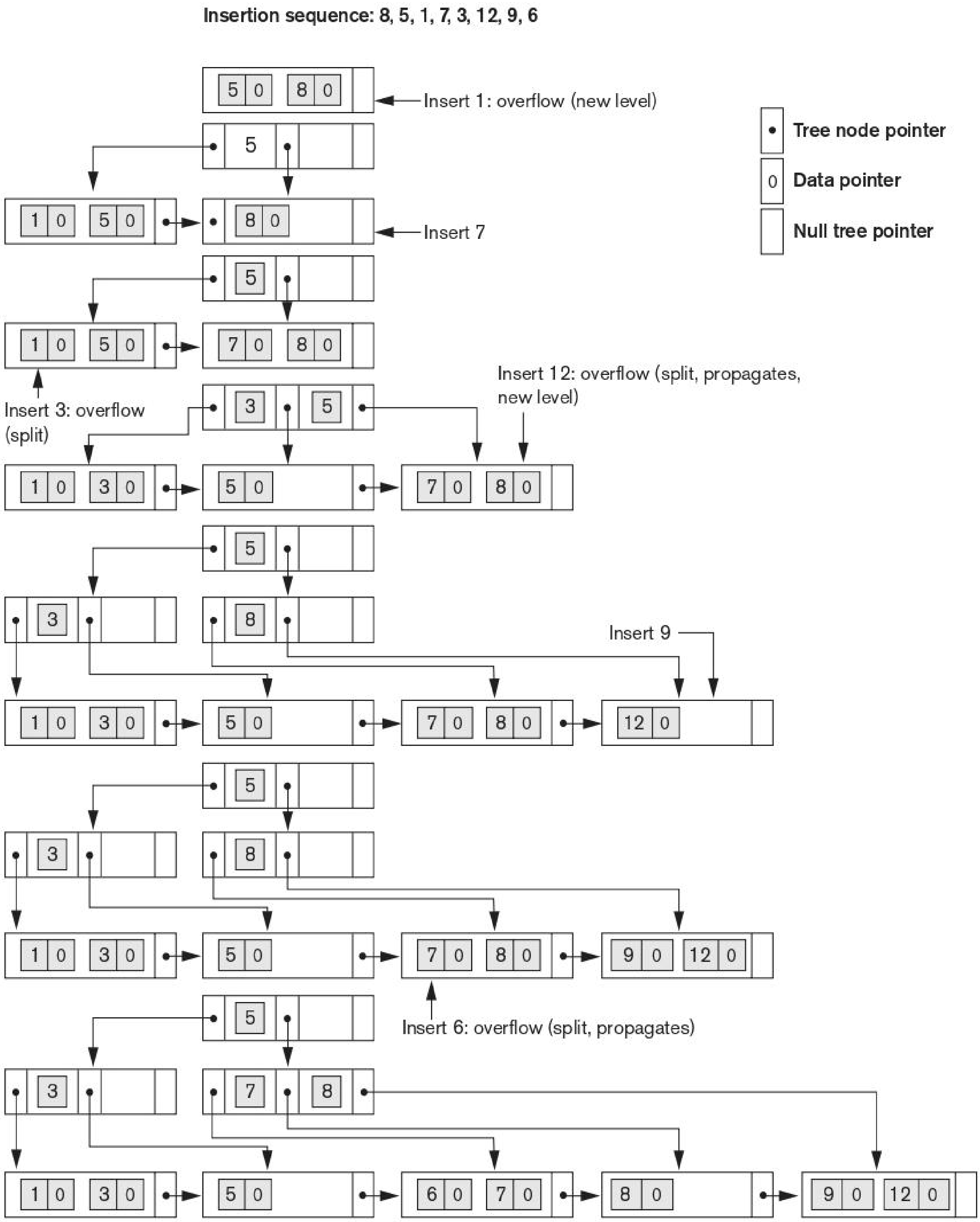
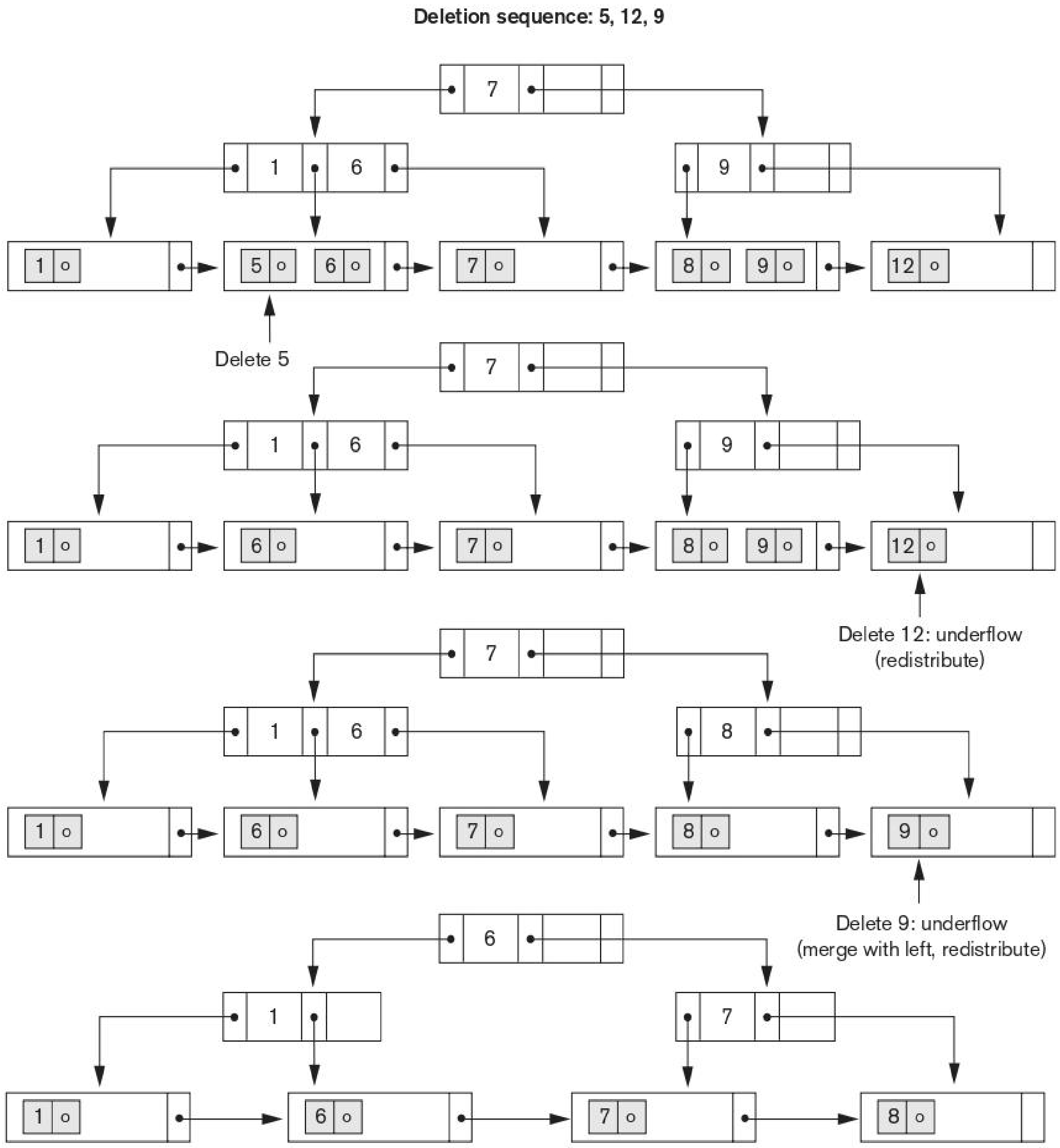

Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6
Insert 1: overflow (new level)
Tree node pointer
Data pointer
Null tree pointer
Insert 7
Insert 3: overflow (split)
Insert 12: overflow (split, propagates, new level)
Insert 9
Insert 6: overflow (split, propagates)

Figure: An example of deletion from a B+-tree



Indexes on Multiple Keys

The primary or secondary keys on which files were accessed were single attributes (fields). In many retrieval and update requests, multiple attributes are involved. If a certain combination of attributes is used frequently.

For example, consider an EMPLOYEE file containing attributes **Dno** (department number), **Age, Street, City, Zip_code, Salary** and **Skill_code**, with the key of **Ssn** (Social Security number).

Consider the query: ***List the employees in department number 4 whose age is 59.*** Note that both **Dno** and **Age** are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 59.

2. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.

3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers. An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions.

All of these alternatives eventually give the correct result.

1. **Ordered Index on Multiple Attributes:**

If we create an index on a search key field that is a combination of <Dno, Age>. The search key is a pair of values <4, 59> in the above example. In general, if an index is created on attributes < A1,A2,…,An>, the search key values are tuples with n values: <v1,v2,…,vn>.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of the department keys for department number 3 precede those for department number 4. Thus <3,n> precedes <4, m>for any values of **m** and **n**. The ascending key order for keys with Dno = 4 would be <4, 18>, <4, 19>, <4, 20> , and so on. Lexicographic ordering works similarly to ordering of character strings.

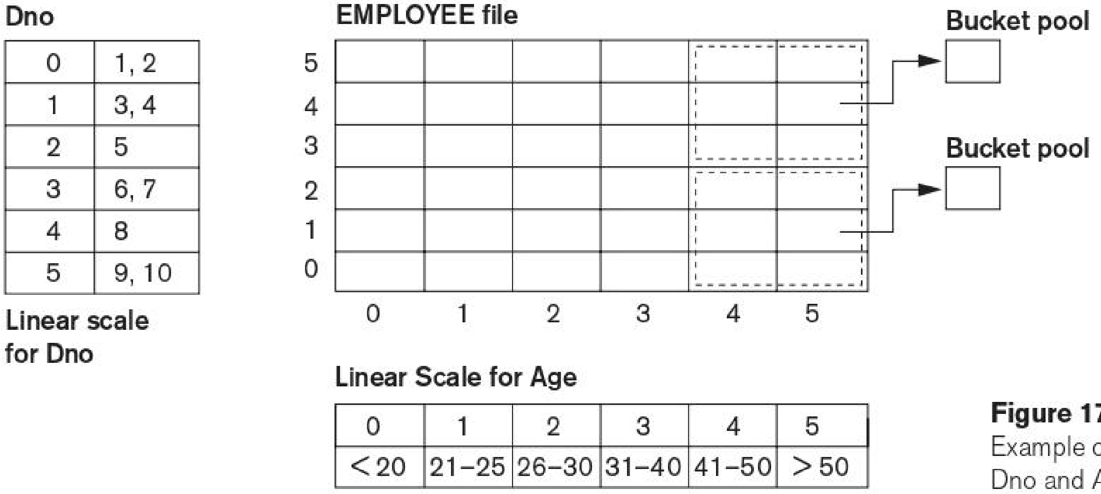
2. Partitioned Hashing:

Partitioned hashing is an extension of static external hashing that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of **n** components, the hash function is designed to produce a result with **n** separate hash addresses. The bucket address is a concatenation of these **n** addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite search key. If **Dno** and **Age** are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that Dno = 4 has a hash address '100' and Age = 59 has hash address '10101'. Then to search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101; just to search for all employees with Age = 59, all buckets (eight of them) will be searched whose addresses are '000 10101', '001 10101', … and so on. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

3. Grid Files:

Another alternative is to organize the EMPLOYEE file as a grid file. If we want to access a file on two keys, say **Dno** and **Age** as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure shows a grid array for the EMPLOYEE file with one linear scale for **Dno** and another for the **Age** attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for Dno has Dno = 1, 2 combined as one value 0 on the scale, whereas Dno = 5 corresponds to the value 2 on that scale. Similarly, Age is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored. Figure also shows the assignment of cells to buckets (only partially).



Dno	
0	1, 2
1	3, 4
2	5
3	6, 7
4	8
5	9, 10

0	1	2	3	4	5
< 20	21–25	26–30	31–40	41–50	> 50

Figure 17.14
Example of a grid array on Dno and Age attributes.

Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. If a range query corresponds to a match on the some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells. **For example**, a query for Dno ≤ 5 and Age > 40 refers to the data in the top bucket shown in Figure.

The grid file concept can be applied to any number of search keys. For example, for n search keys, the grid array would have n dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes.

****

UNIT- III

Transaction Processing

The concept of transaction provides a mechanism for describing logical units of database processing. **Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications.

Introduction to Transaction Processing

In this section, we learn the concepts of concurrent execution of transactions and recovery from transaction failures.

1. Single-User versus Multiuser Systems

A DBMS is **single-user**. One user at a time can use the system.
A DBMS is multiuser. Many users can use the system and access the database **concurrently**. For example, an airline reservations system is used by hundreds of users and travel agents concurrently.

2. Transactions, Database Items, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations. The operations are insertion, deletion, modification or retrieval operations.

One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program.

Read-only transaction:

The operations in a transaction **do not update the database** but **only retrieve data.** Such a transaction is called **a read-only transaction**. Otherwise it is known as a **read-write transaction**.

The basic database access operations are

1. **Read_item(X). Reads a database item named X into a program variable X.**

   Executing a read_item(X) command includes the following steps:
   1. Find the address of the disk block that contains item X.
   2. Copy that disk block into a buffer in main.
   3. Copy item X from the buffer to the program variable named X.

2. **Write_item(X). Writes the value of program variable X into the database item named X.**
   Executing a write_item(X) command includes the following steps:
   1. Find the address of the disk block that contains item X.
   2. Copy that disk block into a buffer in main
   3. Copy item X from the program variable named X into its correct location in the buffer.
   4. Store the updated disk block from the buffer back to disk.

3. **Buffers:** The DBMS will generally generally maintain a number of buffers in main memory that hold database disk block containing the database items being processed.

When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer replacement policy is used. If the chosen buffer has been modified, it must be written back to disk before it is reused.

Transaction and System Concepts

Transaction States and Additional Operations

A transaction is an atomic unit of work that is either completed or not done at all., The recovery manager of the DBMS needs to keep track of the following operations:

■ **BEGIN_TRANSACTION**: This marks the **beginning of transaction** execution.

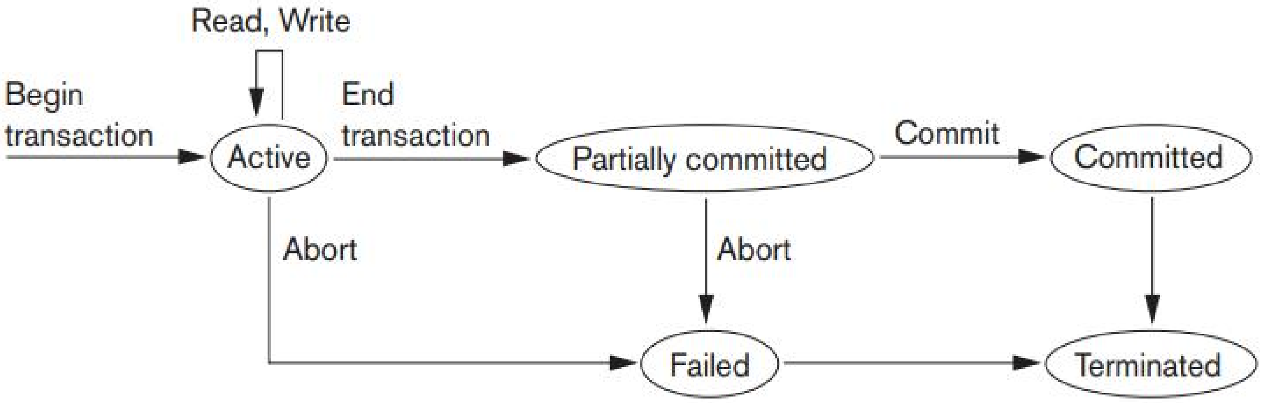
■ **READ or WRITE**: These specify **read or write operations** on the database items.

■ **END_TRANSACTION**: This specifies that READ and WRITE transaction operations **have ended** and marks the end of transaction execution.

■ **COMMIT_TRANSACTION**: This signals a successful end of the transaction so that any changes on the database executed by the transaction can be done.

■ **ROLLBACK (or ABORT):** This signals that the transaction has ended unsuccessfully. So the changes on the database executed by the transaction can be undone.

The following diagram shows the states for transaction execution.



A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially committed state**. At this point, some recovery protocols need, that means recording the changes in the **system log.** After that the transaction reached its commit point and enter the committed state.

Once the **transaction committed**, the transacttion executes successfully and all its changes must be recorded permanently in the database.

A transaction can go to the **failed state,** if one of the checks fails or if the transaction is aborted during its active state. The transaction undo the effect of its WRITE operations on the database.

The System Log

To recovery from failures of transactions, the system maintains a **log** and keep track of all transaction operations that affect the values of database items. The log file contain the following information:

1. [**start_transaction, T**]. Indicates that transaction T has started execution. T refers to a unique transaction-id that is generated automatically by the system for each transaction.
2. [**write_item, T, X, old_value, new_value**]. Indicates that transaction T has changed the value of database item X from old_value to new_value.
3. [**read_item, T, X**]. Indicates that transaction T has read the value of database item X.
4. [**commit, T**]. Indicates that transaction T has completed successfully, and chsnges recorded at the database.
5. [**abort, T**]. Indicates that transaction T has been aborted.

The log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo** the effect of these WRITE operations of a transaction T by using old values of transaction in the log file.

Redoing the operation of a transaction may also be necessary if all its updates are recorded in the log file, all these new_values have been written permanently in the actual database on the disk.

Commit Point of a Transaction

A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be **committed,** and its effect must be permanently recorded in the database.

Desirable Characteristics of Transactions

Transactions should possess several properties, often called the ACID properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

■ **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

■ **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

■ **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

■ **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The **atomicity property** requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database

The **preservation of consistency** is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints. Recall that a database state is a collection of all the stored data items (values) in the database at a given point in time. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold.

The **isolation** property is enforced by the concurrency control subsystem of the DBMS.8 If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks.

The **durability property** is the responsibility of the recovery subsystem of the DBMS. In the next section, we introduce how recovery protocols enforce durability and atomicity.

Levels of Isolation: There have been attempts to define the level of isolation of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads.

Finally, level 3 isolation (also called true isolation) has, in addition to level 2 properties, repeatable reads.9 Another type of isolation is called **snapshot isolation**, and several practical concurrency control methods are based on this. We shall discuss snapshot isolation.

Characterizing Schedules

Characterizing Schedules Based on Recoverability

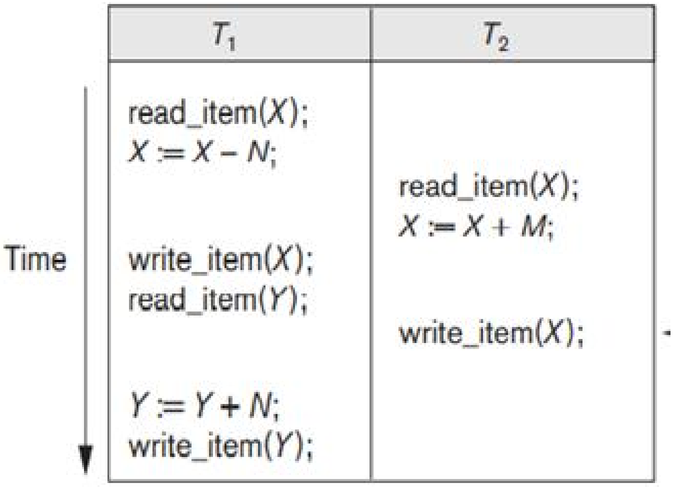
The order of execution of operations from all the various transactions is known as a **schedule (**or **history**).

1. Schedules (Histories) of Transactions:

A **schedule S** of **n** transactions T_1, T_2, … , T_n is an ordering of the operations of the transactions, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i.

A shorthand notation for describing a schedule uses the symbols **b, r, w, e, c**, and **a** for the operations **begin_transaction, read_item, write_item, end_transaction, commit,** and **abort,** respectively.

For ex, the schedule of the following fig. can be written as follows:



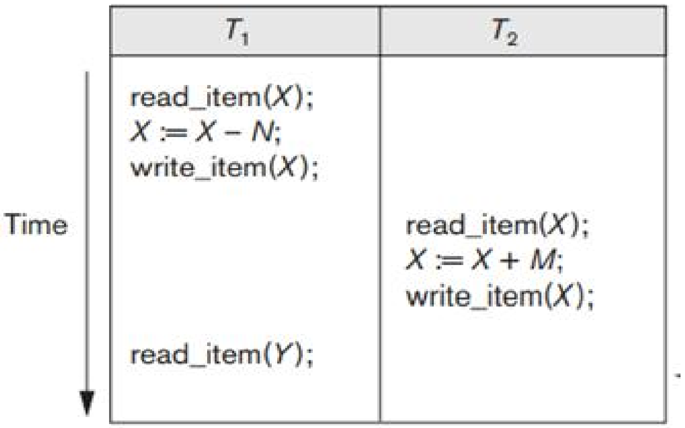
T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

S_a: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);

Similarly the schedule of the following fig. can be written as follows:



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

S_b: r1(X); w1(X); r2(X); w2(X); r1(Y); a1;

Conflicting Operations in a Schedule:

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:

(1) they belong to different transactions;

(2) they access the same data item X; and

(3) at least one of the operations is a write_item(X).

For example, in schedule S_a,

1. the operations r1(X) and w2(X) conflict,
2. the operations r2(X) and w1(X), and
3. the operations w1(X) and w2(X).
4. the operations r1(X) and r2(X) do not conflict, since they are both read operations.
5. the operations w2(X) and w1(Y) do not conflict because they operate on distinct data items X and Y;
6. the operations r1(X) and w1(X) do not conflict because they belong to the sametransaction.

A schedule **S** of **n** transactions T1, T2, … , Tn is said to be a complete schedule if the following conditions hold:

1. The operations in S are exactly those operations in T1, T2, … , Tn, including a commit or abort operation as the last operation for each transaction in the schedule.

2. For any pair of operations from the same transaction Ti, their relative order of appearance in S is the same as their order of appearance in Ti.

2. Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be difficult.

It is important to characterize the types of schedules for which recovery is possible, as well as for which recovery is relatively simple.

A schedule **S** is **recoverable** if no transaction T in S commits until all transactions T′ that have written some item X that T reads have committed.

Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to Sa:

S_a': r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;

S_a' is recoverable, even though it suffers from the lost update problem.

Consider the schedule S_c as follows:

S_c: r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;

S_c is not recoverable because T2 reads item X from T1, but T2 commits before T1 commits.

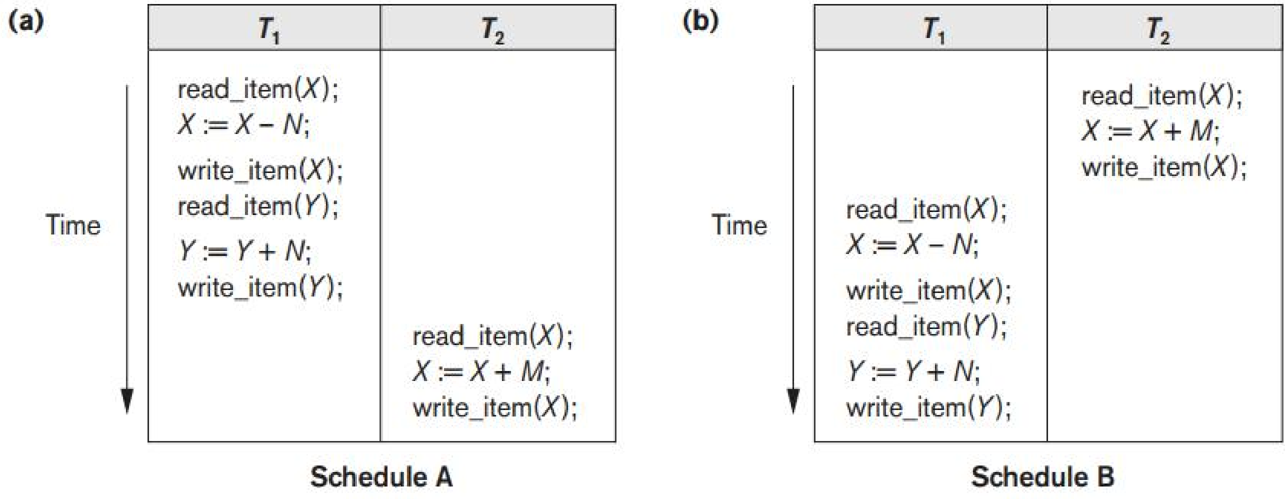
In a recoverable schedule, no committed transaction needs to be rolled back. However, this phenomenon known as cascading rollback to occur, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.

A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions.

Characterizing Schedules Based on Serializability

The types of schedules that are always considered to be correct when concurrent transactions are executing. Such schedules are known as **serializable schedules**. Suppose that two users submit the DBMS transactions T1 and T2 in Figure at the same time.

(a) Serial schedule A: T1 followed by T2. (b) Serial schedule B: T2 followed by T1.



(a)

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

T_1	T_2

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule B

1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).

2. Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).

If interleaving of operations is allowed, many possible orders in which the system can execute the individual operations.

The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

1. Serial, Nonserial, and Conflict-Serializable Schedules

A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called **nonserial.** Therefore, in a serial schedule, only one transaction at a time is active. No interleaving occurs in a serial schedule.

Serial schedule: Entire transactions are performed in serial order: T1 and then T2.

Ex: Schedule A and Schedule B.

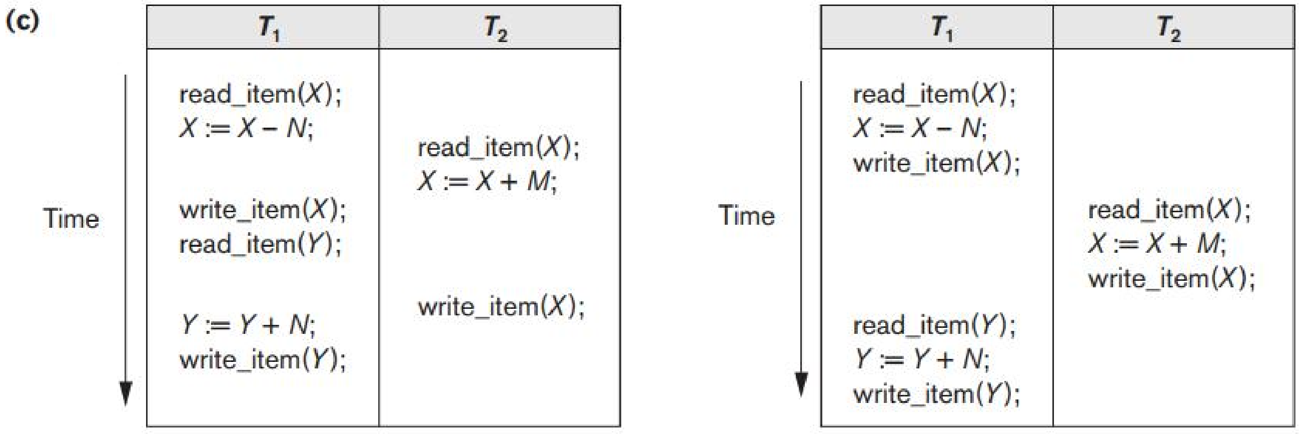
In serial schedule every transaction is executed from beginning to end without any interface from the operations of others transactions, we get a correct end result.

The problem in serial schedules is, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Hence serial schedule are generally considered as unacceptable in practice.

Non-serial schedule: Interleaving the operations of a transactions are called Non-serial schedule.

(c) Two nonserial schedules C and D with interleaving of operations.

(c)



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

Example:

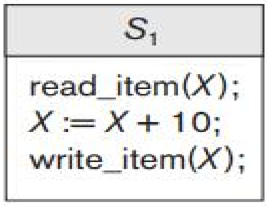
Assume that the initial values of database items are X = 90 and Y = 90 and N = 3 and M = 2.

After executing transactions T1 and T2, expect the database values to be X = 89 and Y = 93 in serial schedules A or B gives the correct results. The non-serial schedule C gives the results X = 92 and Y = 93, in which the X value is erroneous, whereas schedule D gives the correct result.

We would like to determine which of the nonserial schedules always give a correct result and which may give erroneous results.

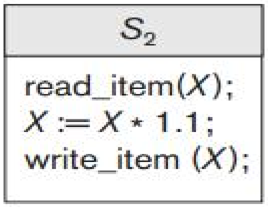
We can form two disjoint groups of the nonserial schedules— those are equivalent to one (or more) of the serial schedules and hence are serializable, and that are not equivalent to any serial schedule and hence are not serializable.

There are several ways to define schedule equivalence. Two schedules are called **result equivalent** if they produce the same final state of the database. For example, in Figure, schedules S1 and S2 will produce the same final database state if they execute on a database with an initial value of X = 100.



S_1

read_item(X);
$X := X + 10$;
write_item(X);



S_2

read_item(X);
$X := X * 1.1$;
write_item (X);

Two definitions of equivalence of schedules are generally used.They are **conflict equivalence** and **view equivalence**.

Conflict Equivalence of Two Schedules: Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules. Otherwise not conflict equivalent.

Example: S1: r1(X), w2(X)

S2: w2(X), r1(X) not conflict equivalent.

Serializable Schedules: A schedule is to the conflict serializable if it is equivalent to some serial schedule S′. In such a case, we can reorder the nonconflicting operations in S until we form the equivalent serial schedule S′.

2. Testing for Serializability of a Schedule

There is a simple algorithm for determining the conflict serializablility of a schedule.

The algorithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph**, which is a **directed graph** G = (N, E) that consists of a set of nodes N = {T1, T2, … , Tn } and a set of directed edges E = {e1, e2, … , em }. There is one node in the graph for each transaction Ti in the schedule. Each edge ei in the graph is of the form (Tj → Tk), 1 ≤ j ≤ n, 1 ≤ k ≤ n, where Tj is the **starting node** and Tk is the **ending node** of ei. Such an edge created if one of the operation in Tj appears in the schedule before the conflicting operation in Tk.

Algorithm: Testing Conflict Serializability of a Schedule S

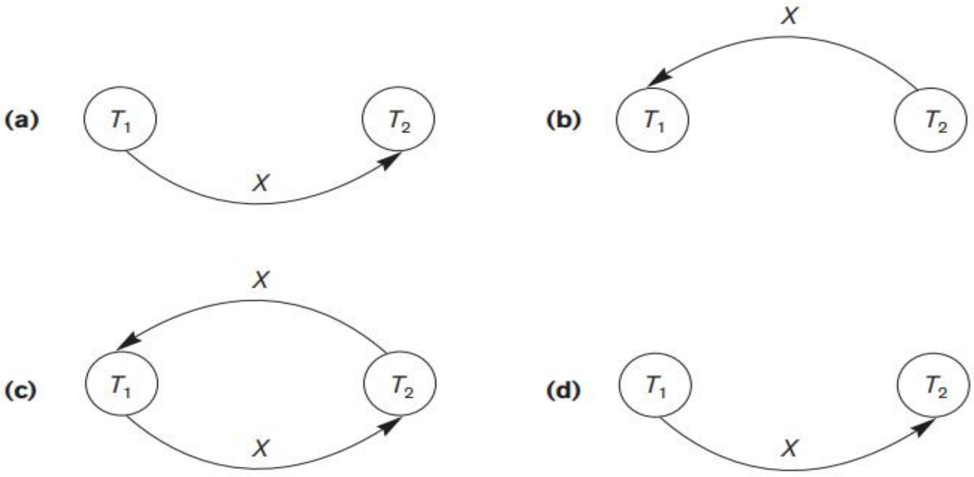
1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
2. For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) in the precedence graph.
3. For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create an edge (Ti → Tj) in the precedence graph.
4. For each case in S where Tj executes a write_item(X) after Ti executes a

write_item(X), create an edge (Ti → Tj) in the precedence graph

5. The schedule S is serializable if and only if the precedence graph has no cycles.

In general, several serial schedules can be equivalent to **S** if the precedence graph for **S** has no cycle. However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so **S** is not serializable.

Figure: (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).



3. **View Equivalence and View Serializability**

Two schedules S and S′ are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S′, and S and S′ include the same operations of those transactions.
2. For any operation ri(X) of Ti in S, if the value of X read by the operation has been written by an operation wj(X) of Tj, the same condition must hold for the value of X read by operation ri(X) of Ti in S′.
3. If the operation wk(Y) of Tk is the last operation to write item Y in S, then wk(Y) of Tk must also be the last operation to write item Y in S′.

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.

The condition 3 ensures that the final write operation on each data item is the same in both schedules. A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

Concurrency Control Techniques

There are two techniques used to control the concurrency, They are

1. Looking
2. Time stamps

Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item.

Binary Locks: A binary lock can have two states. They are locked(1), unlocked (0).

If the value of the **lock on X is 1**, then item X cannot be accessed by other requests. If the value of the **lock on X is 0**, then item can be accessed by other transactions.

In simple binarysequence every transaction must obey the following rules:

1. A transaction T must issue the operation **lock_item(X)** before **any read_item(X)** or **write_item(X)** operations are performed in T.
2. A transaction T must issue the operation **unlock_item(X)** after all **read_item(X)** and **write_item(X)** operations are completed in T.
3. A transaction T will **not issue a lock_item(X**) operation if it already holds the lock on item X.
4. A transaction T will not issue an **unlock_item(X)** operation unless it already holds the lock on item X.

Shared/Exclusive (or Read/Write) Locks:

If a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock, called a **multiple-mode lock**, is used. In this scheme—called **shared/exclusive or read/write locks.**

A lock associated with an item X, LOCK(X) has three possible states:

1. Read-locked
2. Write-locked
3. Unlocked

A **read-locked** item is also called **share-locked** because other transactions are allowed to read the item, whereas **write-locked** item is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

When we use the shared ro exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation **read_lock(X) or write_lock(X**) before any **read_item(X)** operation is performed in T.
2. A transaction T must issue the operation **write_lock(X)** before any **write_item(X)** operation is performed in T.
3. A transaction T must issue the operation **unlock(X)** after all **read_item(X)** and **write_item(X)** operations are completed in T.
4. A transaction T will **not issue a read_lock(X)** operation if it already holds a read lock or a write lock on item X.
5. A transaction T will **not issue a write_lock(X)** operation if it already holds a read lock or write lock on item X.
6. A transaction T will **not issue an unlock(X)** operation unless it already holds a read lock or a write lock on item X.

Conversion (Upgrading, Downgrading) of Locks.

A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.

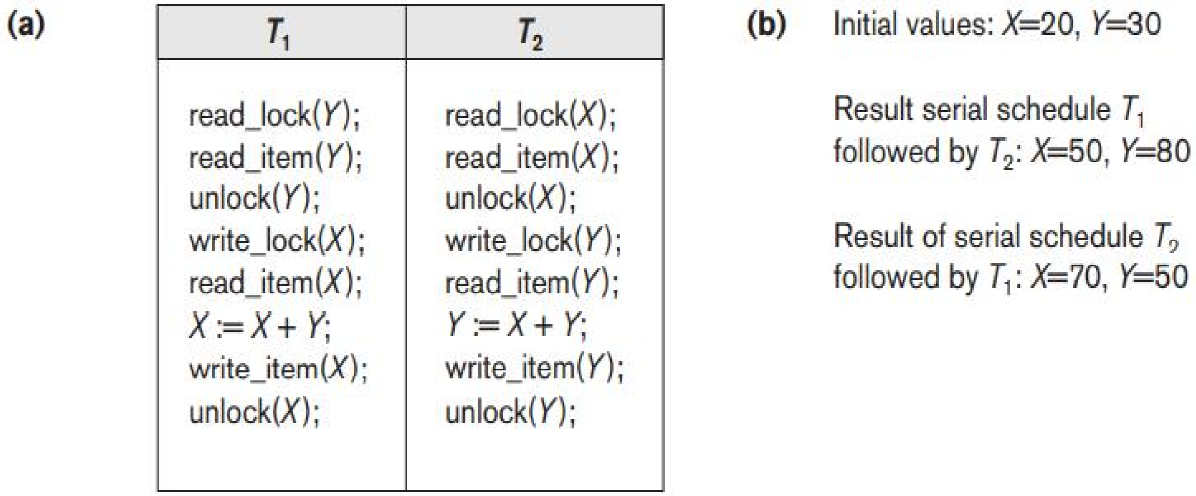
For example, a transaction T issue a read_lock(X) and then later to **upgrade the lock** by issuing a write_lock(X) operation.

For example, a transaction T to issue write_lock(X) and then later to **downgrade the lock** by issuing a read_lock(X) operation.

Guaranteeing Serializability by Two-Phase Locking:

1. Using binary locks or read or write locks in transaction does not **guarantee serializability** of schedules.
2. The following example show the preceding locking rules are followed but a non serial schedule will gives the wrong result.

Seriaal:



(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

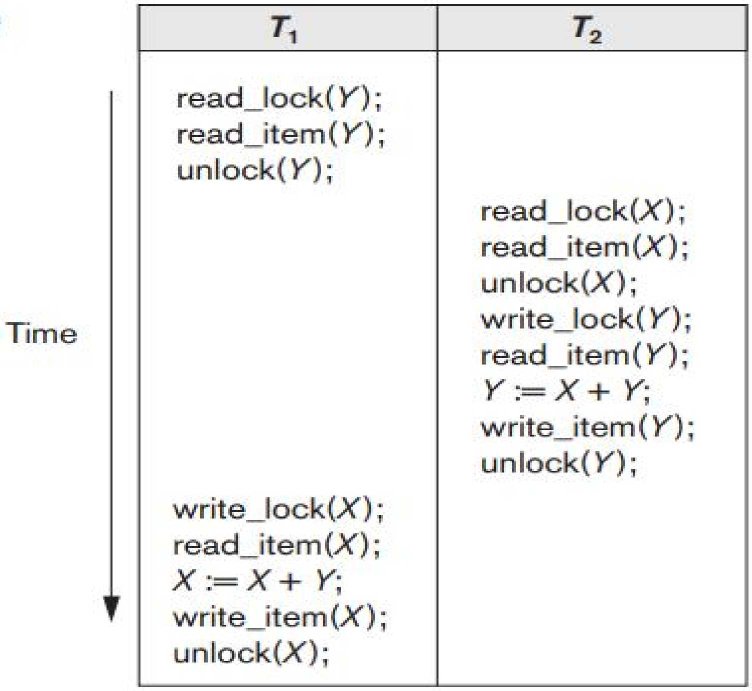
(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

This is because in serializable schedule the Y in T1 was unlocked too. It will gives correct result, we must follow an additional protocol. The best known protocol is Two Phase Locking.

Non Serial:



T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Result of schedule S: X=50, Y=50 (nonserializable)

A transaction is said to follow the two phase locking protocol. If all locking operations preceed the first unlock operation in the transaction.

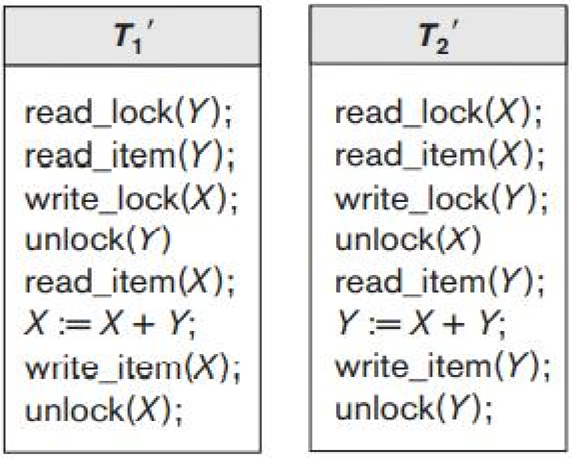
In the two phase locking a transaction can be divided into 2 phases. They are **growing phase, shinking phase.**

Growing or Expanding Phase: In growing phase new locks on items can be acquired but not release the locks.

Shinking Phase: In this locks can be released but no new locks are acquired.

The Transactions T1 and T2 in Figure (a) do not follow the two-phase locking. Because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2.

If we follow two phase locking the transactions can be rewritten as T1′ and T2′ as follows:



T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

It can be proved that in every transaction in a schedule follow the two phase locking protocol, the schedule is guaranteed to be serializible.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking.

There are a number of variations of two-phase locking (2PL).

Conservative 2 Phase Locking: In this it requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set. Conservative 2PL is a deadlock-free protocol.

Static 2 Phase Locking: In this a transaction T does not release any of its write locks until after it commits (or) aborted, it is not a deadlock free.

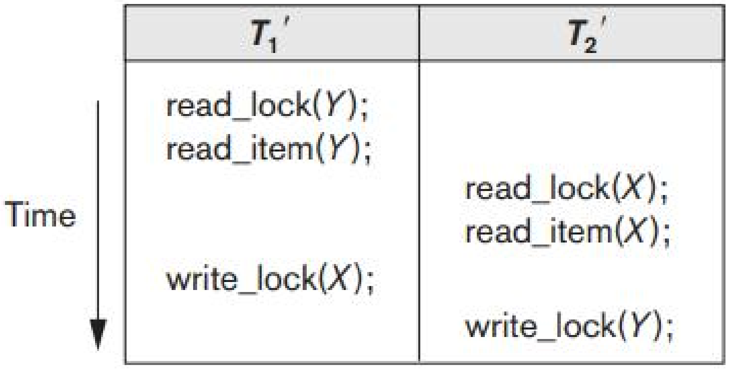
Rigorous 2Phase Locking: In this a transaction T does not release any of its locks (read lock or write lock) until after it commits or aborts.

The two phase locking protocol guarentees serializability, but the use of locks can cause two additional problems. They are

1. Dead lock   2. Starvation

Dead Lock: The Transaction T_1' acquires a lock on database item(y). The transaction T_2' acquires a lock on database item(x). The transation T_1' needs database item X to complete their work and the transaction T_2' nedds database item Y. The two transactions waiting for the database items locked by other transactions.

Figure: A partial schedule of T1′ and T2′ that is in a state of deadlock.



T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Deadlock Prevention Protocols:

In **prevent deadlock** to use a deadlock prevention protocol. One of the deadlock prevention protocol, which is conservative in two-phase locking, requires that every transaction lock all the items it needs in advance. If any of the items cannot be obtained, none of the items are locked. The transaction tries to lock the data item.

A number of deadlock prevention schemes have been proposed that make a decision which transaction should be wait and which transaction should be aborted. These techniques can use the concept of **transaction timestamp** TS(T).

The timestamps are typically based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2). The older transaction has smaller time stamp value.

Two schemes that prevent deadlock are called wait-die and wound-wait.

The rules followed by these schemes are:

■ **Wait-die:** If TS(Ti) < TS(Tj), then (Ti older than Tj) Ti is allowed to wait; otherwise (Ti younger than Tj) abort Ti (Ti dies) and restart it later with the same timestamp.

■ **Wound-wait:** If TS(Ti) < TS(Tj), then (Ti older than Tj) abort Tj (Ti wounds Tj) and restart it later with the same timestamp; otherwise (Ti younger than Tj) Ti is allowed to wait.

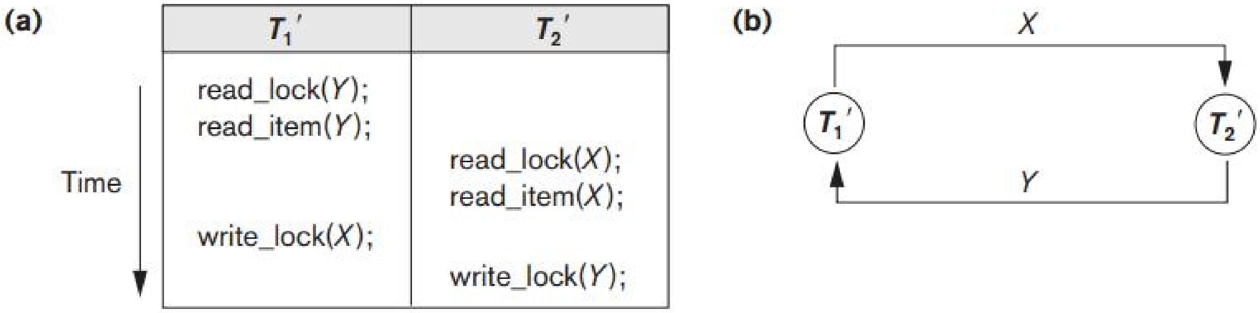
Deadlock Detection:

A simple way to detect deadlock is to construct a wait for graph for eah transaction is currently executed.

Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj , a directed edge (Ti → Tj) is created in the wait-for graph.

Whenever a transaction Tj is waiting to lock an item Y that is currently locked by a transaction Tj , a directed edge (Tj → Ti) is created in the wait-for graph.

We have a state of deadlock if and only if the wait-for graph has a cycle.



T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

If the system is in a state of deadlock, choosing which transactions to abort is known as **victim selection**. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should select transactions that have not made many changes.

Time outs: If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts.

Concurrency Control Based on Timestamp Ordering

A **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system. We will refer to the timestamp of transaction T as TS(T).

Timestamp Ordering Algorithm:

In this scheme order of transactions are based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called **timestamp ordering (TO).**

In this algorithm each database item X has two timestamp (TS) values:

1. read_TS(X): The **read timestamp** of item X is the largest timestamp of transactions that have successfully read item X.

i.e., read_TS(X) = TS(T), where T is the youngest transaction that has read X successfully.

2. write_TS(X): The **write timestamp** of item X is the largest timestamps among all time stamps of transactions that have successfully written item X.

i.e., write_TS(X) = TS(T), where T is the youngest transaction that has written X successfully.

Basic Timestamp Ordering (TO):

The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Transaction T issues a **write_item(X)** operation, the following check is performed:

a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation.

This should be done because some younger transaction read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2. Transaction T issues a **read_item(X)** operation, the following check is performed:

a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation.

This should be done because some younger transaction already written the value of item X before T had a chance to read X.

b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Strict Timestamp Ordering (TO):

A transaction T issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction has committed or aborted.

Thomas's Write Rule:

A modification of the basic TO algorithm, known as Thomas's write rule.

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.

2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. We must ignore the write_item(X) operation of T because it is already outdated any conflict arising the situation would be detected by case (1).

3. If neither the condition in case (1) nor the condition in case (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

Validation Concurrency Control

In **optimistic concurrency control** techniques, also known as **validation** or **certification techniques**, no checking is done while the transaction is executing.

In the transaction execution updates are not applicable directly to the database items until the transaction reaches the end.

During transaction execution, all updates are applied to local copies of the data items. At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability.

If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted.

There are three phases for this concurrency control protocol:

1. Read phase
2. Validation phase
3. Write phase

1. **Read phase:**

A transaction can read values of committed data items from the database.

2. **Validation phase**:

Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.

In validation phase if one of the condition holds on the transactions, the transaction is valid, otherwise the transaction is not valid.

1. Transaction Tj completes its write phase before Ti starts its read phase.

2. Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj.

3. Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase before Ti completes its read phase.

3. **Write phase:**

If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

■ A database record

■ A field value of a database record

■ A disk block

■ A whole file

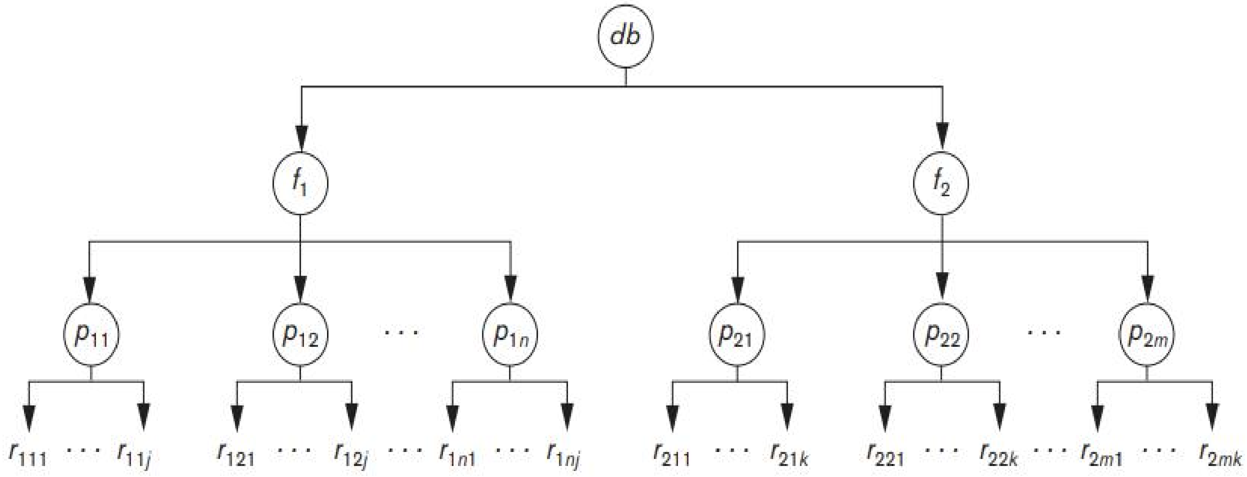
■ The whole database

The size of data items is often called the **data item granularity**. ***Fine granularity*** refers to small item sizes, whereas ***coarse granularity*** refers to large item sizes.

Multiple Granularity Level Locking

The following diagram shows a simple granularity hierarchy with a database containing two files. Each file containing several disk pages, and each page containing several records. This can be used to illustrate a **multiple granularity** level 2PL protocol.

Figure: A granularity hierarchy for illustrating multiple granularity level locking.


db
f1
f2
p11
p12
. . .
p1n
p21
p22
. . .
p2m
r111 . . . r11j
r121 . . . r12j . . . r1n1 . . . r1nj
r211 . . . r21k
r221 . . . r22k . . . r2m1 . . . r2mk

Suppose transaction T1 wants to update all the records in file f1, and T1 requests and is granted an exclusive lock for f1. Then all of f1's pages (p11 through p1n)and the records contained on those pages are locked in exclusive mode.

Suppose another transaction T2 only wants to read record r_{1nj} from page p_{1n} of file f1; then T2 would request a shared record-level lock on r_{1nj}. However, the database system must verify the requested lock with already held locks. One way to verify this is to traverse the tree from the leaf r_{1nj} to p_{1n} to f1 to db. If at any time a conflicting lock is held on any of those items, then the lock request for r1nj is denied and T2 is blocked and must wait. This traversal would be fairly efficient.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed.

Intention locks: The intention locks of a transaction to indicate along the path from the root to the desired node, There are three types of intention locks:

1. **Intention-shared (IS):** It indicates that one or more shared locks will be requested on some descendant node(s).

2. **Intention-exclusive (IX):** It indicates that one or more exclusive locks will be requested on some descendant node(s).

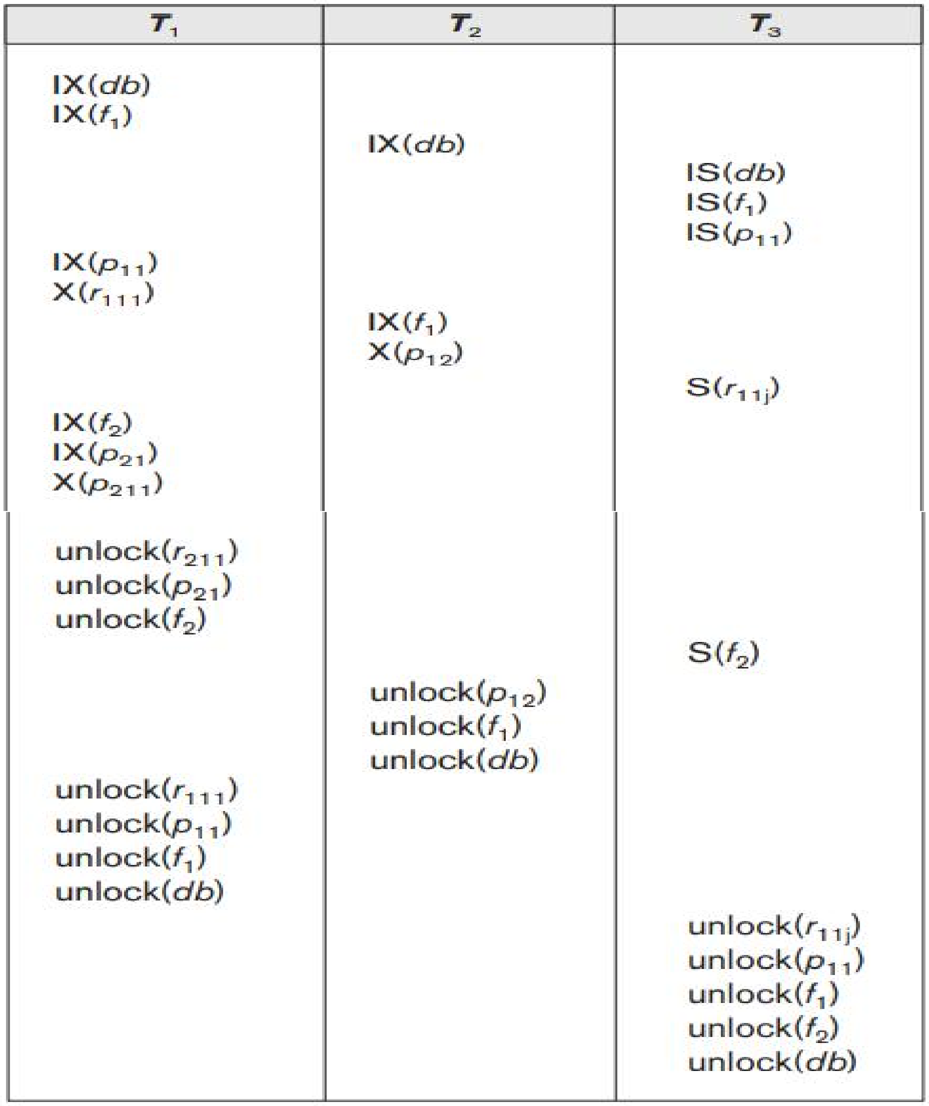
3. **Shared-intention-exclusive (SIX):** It indicates that the current node is locked in shared mode but one or more exclusive locks will be requested on some descendant node(s).

The multiple granularity locking (MGL) protocol consists of the following rules:

1. The root of the tree must be locked first, in any mode.
2. A node N locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.
3. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
4. A transaction T can lock a node only if it has not unlocked any node.
5. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

Example: Consider the following three transactions:

1. T1 wants to update record r_{111} and record r_{211}.
2. T2 wants to update all records on page p_{12}.
3. T3 wants to read record r_{11j} and the entire f2 file.



T_1	T_2	T_3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(db)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(db)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(db)

Database Recovery Techniques

Database Recovery Concepts

1.Recovery Outline and Categorization of Recovery Algorithms:

Recovery from transaction failures usually means that the database is restored to the most recent consistent state before the time of failure. This information is typically kept in the **system log**. A typical strategy for recovery may be summarized informally as follows:

1. Due to **catastrophic** failure, such as a disk crash, redoing the operations.
2. When the database is not physically damaged, but has become inconsistent due to **non-catastrophic** failure by undoing some operations.

Two main techniques for recovery from non-catastrophic transaction faillures are

1. **Deferred update** or NO-UNDO/REDO algorithm.
2. **Immediate update** or UNDO/REDO algorithm.

The **Deferred update** techniques do not **physically update** the database on disk until after a transaction commits; then the updates are recorded in the database. If a transaction fails before reaching its commit point, it will not have changed the database on disk in any way. So UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk. Hence, deferred update is also known as the **NO-UNDO/REDO** algorithm.

In the **immediate update** techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point. These operations are typically recorded in the log on disk by force-writing before they are applied to the database. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both undo and redo may be required during recovery. This technique, known as the **UNDO/REDO** algorithm, requires both operations.

2.Caching (Buffering) of Disk Blocks:

Typically, multiple disk pages that include the data items to be updated are **cached** into main memory buffers and then updated in memory before being written back to disk.

A collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding these buffers. A **directory** for the cache is used to keep track of which database items are in the buffers. This can be a table of < Disk_page_address, Buffer_location,…> entries.

When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is not, the item must be located on disk, and the appropriate disk pages are copied into the cache.

It may be necessary to replace (or flush) some of the cache buffers to make space available for the new item. Some page replacement strategy from OS such as LRU, FIFO can be used to select burrers for replacement.

Associated with each buffer in the cache is a **dirty bit**, which can be included in the directory entry to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, the cache directory with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). When the buffer contents are replaced from the cache, the contents must first be written back to the corresponding disk page only if its dirty bit is 1.

Two main strategies can be employed when replacing a modified buffer back to disk. The first strategy, known as in**-place updating**, writes the buffer to the same original disk location, thus overwriting the old value of any changed data items on disk. Hence, a single copy of each database disk block is maintained. The second strategy, known as **shadowing**, writes an updated buffer at a different disk location, so multiple versions of data items can be maintained.

In general, the old value of the data item before updating is called the **before image (BFIM),** and the new value after updating is called the **after image (AFIM**). If shadowing is used, both the BFIM and the AFIM can be kept on disk.

3.Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

Write-Ahead Logging: In this protocol, the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk.

Steal/No-steal:

No-Steal: If a cache buffer page updated by a transaction cannot be written to disk before the transaction commits, the recovery method is called a no-steal approach.

Steal: if the recovery protocol allows writing an updated buffer before the transaction commits, it is called steal.

Force/No-force: If all pages updated by a transaction are immediately written to disk before the transaction commits, the recovery approach is called a force approach. Otherwise, it is called no-force.

4.Checkpoints in the System Log and Fuzzy Checkpointing:

Checkpoints in the System Log: Another type of entry in the log is called a checkpoint. The checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.

2. Force-write all main memory buffers that have been modified to disk.

3. Write a [checkpoint] record to the log, and force-write the log to disk.

4. Resume executing transactions.

Fuzzy Checkpointing: To reduce this delay, it is common to use a technique called fuzzy check point.

In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, …] record is written in the log with the relevant information collected during checkpointing. The system maintains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

5.Transaction Rollback and Cascading Rollback:

Transaction Rollback: If a transaction fails for whatever reason after updating the database, it may be necessary to **roll back** the transaction.

If a transaction **T** is rolled back, any transaction **S** that has, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S must also be rolled back; and so on. This phenomenon is called **cascading rollback**, and it can occur when the recovery protocol ensures recoverable schedules but does not ensure strict or cascadeless schedules.

Understandably, cascading rollback can be complex and time-consuming. That is why almost all recovery mechanisms are designed so that cascading rollback is never required.

Recovery Based on Deferred Update

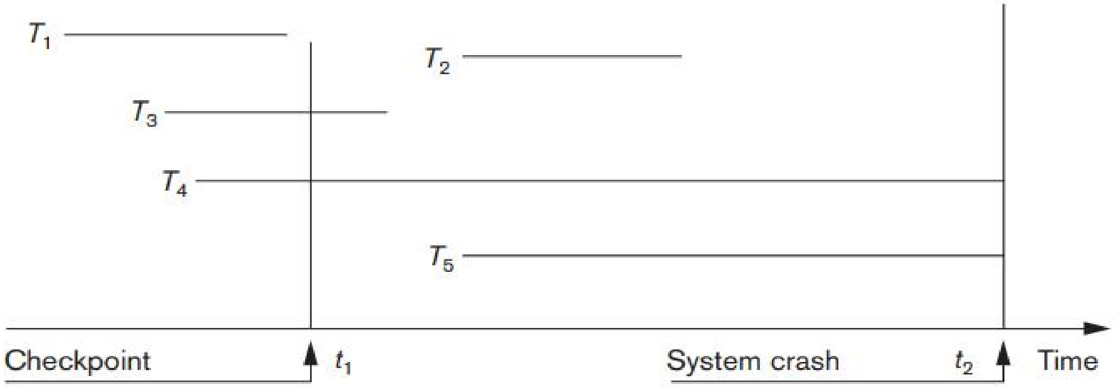
The deferred update thechniques do not physically update the database on disk until after a transaction reaches its commit point, then the updates are recorded in the database. It will not have changed the database in any way, so UNDO is not needed. It may be necesay to REDO the effect of the operation.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point.
2. A transaction does not reach its commit point until all its update operations are recorded in the log and the log is force-written to disk.

Notice that step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. Hence, this is known as the NO-UNDO/REDO recovery algorithm. REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk.

Figure: An example of a recovery timeline to illustrate the effect of checkpointing.



When the checkpoint was taken at time t1, transaction T1 had committed, whereas transactions T3 and T4 had not. Before the system crash at time t2, T3 and T2 were committed but not T4 and T5.

According to the RDU_M method, there is no need to redo the write_item operations of transaction T1—or any transactions committed before the last checkpoint time t1.

The write_item operations of T2 and T3 must be redone, however, because both transactions reached their commit points after the last checkpoint.

Transactions T4 and T5 are rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol.

Recovery using Deferred Update in a Single-User Environment:

Procedure RDU-S: use two lists of transactions:

The committed transactions since the last check point, and the active transactions. Apply the REDO operations to all the write_item operations of the committed transactions from the log in the order in which they written to the log. Restart the active transactions.

The REDO procedure is defined as follows:

Procedure REDO (WRITE_OP): Redoing a write_item operation WRITE_OP consists of examining its log entry [write_item, T, X, new_value] and setting the value of item X in the database to new_value, which is the after image (AFIM).

Deferred Update with Concurrent Execution in a Multi user Environment:

Procedure RDU_M (NO-UNDO/REDO with checkpoints): Use two lists of transactions maintained by the system; the committed transactions T since the last checkpoint (commit list), and the active transactions T′ (active list). REDO all the WRITE operations of the committed transactions from the log, in the order in which they were written into the log. The transactions that are active and did not commit are effectively canceled and must be resubmitted.

Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated immediately, without any need to wait for the transaction to reach its commit point.

UNDO/REDO Recovery based on Immediate Update in a Single_User Environment:

Procedure RIU_S:

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions (atmost, one transactionwill fall in this category because the sysem is single-user).

2. Undo all the write_item operations of the active transaction from the log, using the UNDO procedure described below.

3. Redo the write_item operations of the committed transactions from the log, in the order in which they were written into the log, using the REDO procedure defined earlier.

UNDO/REDO Recovery based on Immediate Update with current execution:

Procedure RIU_M:

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.

2. Undo all the write_item operations of the active (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.

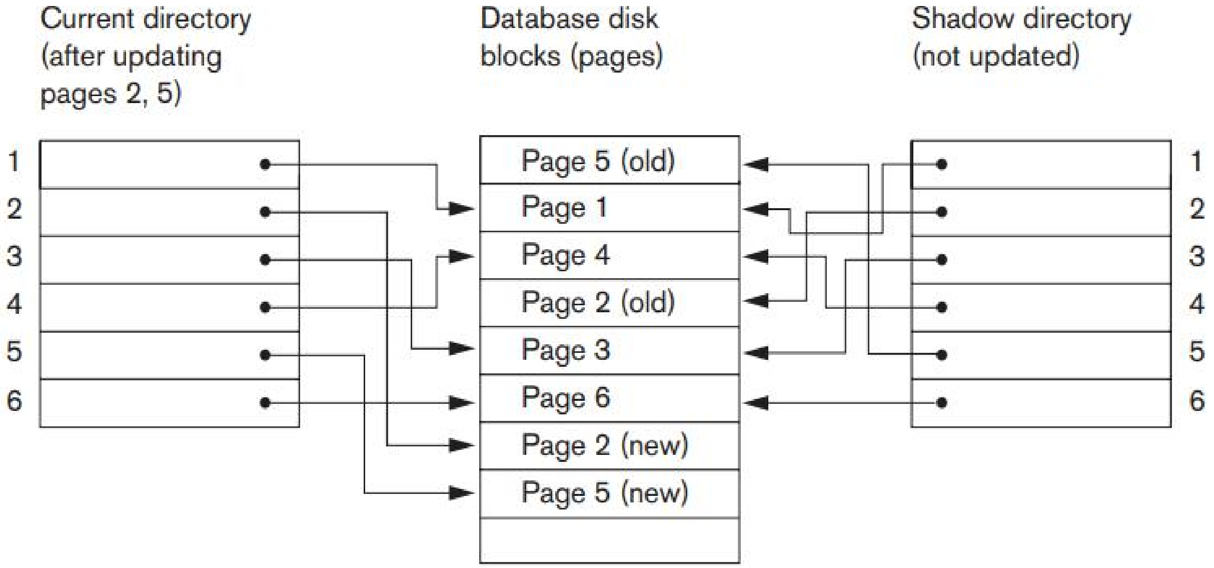
3. Redo all the write_item operations of the committed transactions from the log, in the order in which they were written into the log.

Shadow Paging

Shadow paging considers the database to be made up of a number of fixedsize disk pages (or disk blocks)—say, **n**—for recovery purposes. A directory with **n** entries is constructed, where the i^{th} entry points to the i^{th} database page on disk.

The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

Figure: An example of shadow paging.


Current directory (after updating pages 2, 5)
Database disk blocks (pages)
Shadow directory (not updated)
1
2
3
4
5
6
Page 5 (old)
Page 1
Page 4
Page 2 (old)
Page 3
Page 6
Page 2 (new)
Page 5 (new)
1
2
3
4
5
6

During transaction execution, the shadow directory is never modified. When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. The current directory entry is modified to point to the new disk block.

From the above Figure illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

To recover from a failure during transaction execution, it is sufficient to discard the current directory and reinstating the shadow directory. Thus the database returns to its state prior to the transaction execution. Committing a transaction corresponds to discarding the previous shadow directory.

The ARIES Recovery Algorithm

The ARIES recovery procedure consists of 3 main steps.

1. Analysis
2. REDO
3. UNDO

1. **Analysis:** Identifies the dirty pages in the buffer and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined.

2. **REDO:** The REDO operation is applied only to committed transactions. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached.

3. **UNDO:** The log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order.

Log sequence number (LSN): Each log record has an associated log sequence number in incremented, indicate the address of the log record on disk.

Previous LSN: Each log record has associated previous LSN for that transaction.

Two tables are needed for efficient recovery,

1. Transaction table
2. Dirty page table

Which are maintained by the transaction manager. When a crach occurs then table are build in the analysis phase of recovery.

After a crach, the ARIES recovery manager take over information.

Analysis Phase: The analysis phase starts at the begin_checkpoint record and proceeds to the end of the log. When the end_checkpoint record is encountered, the Transaction Table and Dirty Page Table are accessed. During analysis, the log records being analyzed may cause modifications to these two tables. After the check point in the system log, each transaction is compared with transaction table entries, if it is not in that transaction add to the transaction table, already exist change Last_LSN to LSN in the log.

REDO phase: Find the smallest LSN, M of all the dirty pages in the dirty page thable, which indicate the log position where ARIES ready to start REDO phase.

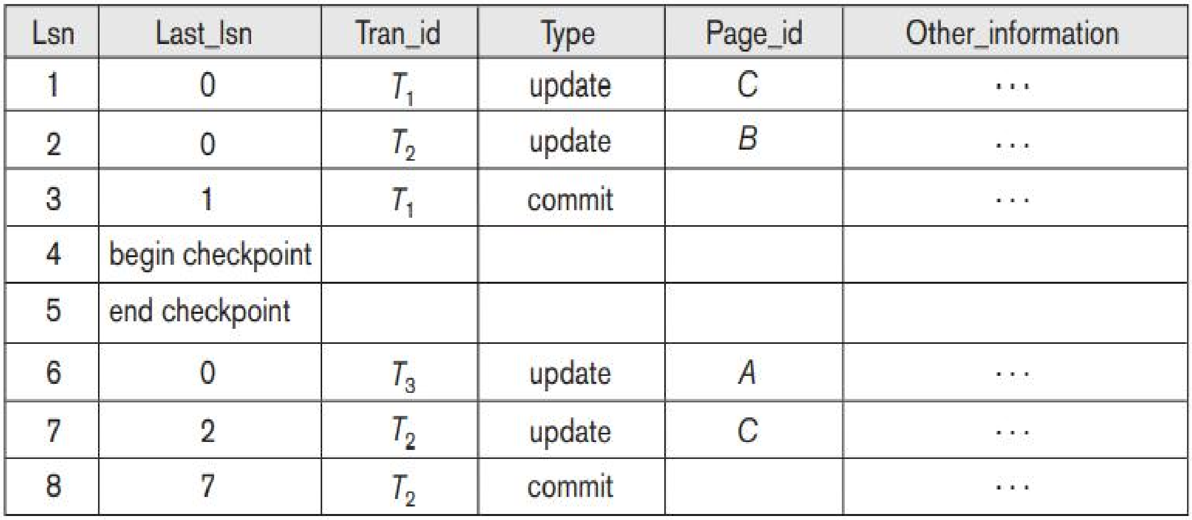
The REDO start at the log record with LSN=M and scans forward to the end of log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. Once the REDO phase is finished, the database come prior of the failure.

UNDO pahse: The set of active transactions called the undo set identified in the transaction table during the analysis phase. Now undo phase proceeds by scanning bcakward from end of the log and undoing the appropriate actions. When this is completed, the recovery process is finished.

Ex: There are 3 transactions T_1,T_2,T_3.

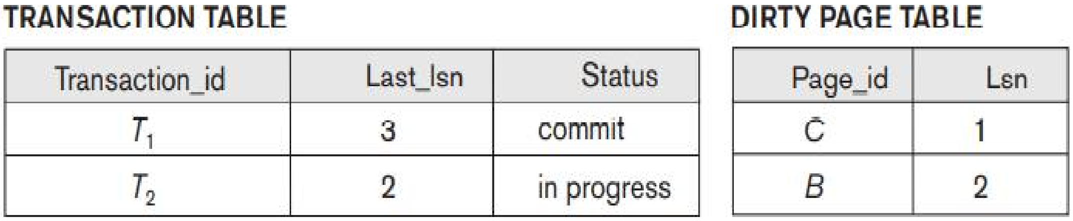
Figure: An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

(a)



Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	. . .
2	0	T_2	update	B	. . .
3	1	T_1	commit		. . .
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	. . .
7	2	T_2	update	C	. . .
8	7	T_2	commit		. . .

(b)



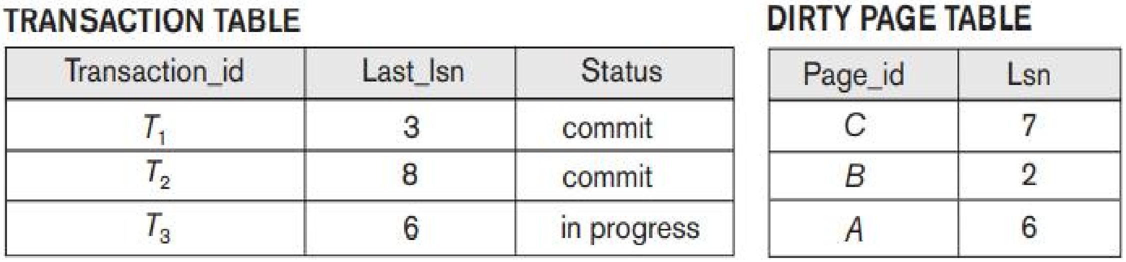
TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)



TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	7
B	2
A	6

Suppose that a crash occurs at this point, the address associated begin_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end.

The end_checkpoint record contains the Transaction Table and Dirty Page Table in Figure (b), and the analysis phase will further reconstruct these table as shown in Figure (c).

When the analysis phase log record 6, a new entry for transaction T3 is made in the Transaction Table and a new entry for page A is made in the Dirty Page Table.

After log record 8 is analyzed, the status of transaction T2 is changed to committed in the Transaction Table.

For the REDO phase, the smallest LSN in the Dirty Page Table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. In our ex, the pages C,B,A will be read agin and the updates reapplied from the log. The REDO phase completed.

Now the UNDO phase stars from the transactiontable, UNDO is applied only to the active transaction T3. The UNDO phase starts at log entry 6 and proceeds backward in the log.

****

UNIT- IV

Object-Relational Systems

Object Relational Supposrt in SQL

SQL as the standard language for RDBMSs. SQL was first specified by Chamberlin and Boyce (1974) and underwent enhancements and standardization in 1989 and 1992

The relational model with object database enhancements is sometimes referred to as the object-relational model.

The following are some of the object database features that have been included in SQL:

■ Some **type constructors** have been added to specify complex objects. These include the row type, which corresponds to the tuple (or struct) constructor. An array type for specifying collections is also provided.

■ A mechanism for specifying **object identity** through the use of reference type is included.

■ **Encapsulation of operations** is provided through the mechanism of user-defined types (UDTs) that may include operations as part of their declaration. These are somewhat similar to the concept of abstract data types that were developed in programming languages

■ **Inheritance** mechanisms are provided using the keyword UNDER.

Type Constructures:

Type Constructures row and array are used to specify complex types. These are also known as **user-defined types (UDTs).** A UDT may be specified in its simplest form using the following syntax:

CREATE TYPE TYPE_NAME **AS** [ROW] (< component declarations>);

The keyword ROW is optional.

An example for specifying a row type for addresses and employees may be done as follows:

```
    CREATE TYPE   ADDR_TYPE   AS (
        STREET_ADDR     ROW ( NUMBER VARCHAR (5),
                              STREET_NAME  VARCHAR (25),
                              APT_NO VARCHAR (5),
                              CITY VARCHAR (25),
                              ZIP VARCHAR (10)
                            );

   CREATE TYPE  EMP_TYPE   AS ( NAME VARCHAR    (35),
                                ADDR ADDR_TYPE,
                                AGE INTEGER );
```

Object Identifiers Using Reference Types

Unique system-generated object identifiers can be created via the reference type using the keyword REF.

In general, the user can specify that system-generated object identifiers for the individual rows in a table should be created. By using the syntax:

```
    REF IS  < OID_ATTRIBUTE> < VALUE_GENERATION_METHOD>;
```

For example, we can creat two tables based on the row type declarations given earlier as follows:

```
  CREATE TABLE EMPLOYEE OF EMP_TYPE REF IS EMP_ID
  SYSTEM    GENERATED;

  CREATE TABLE COMPANY OF COMP_TYPE (
  REF IS COMP_ID  SYSTEM GENERATED,
  PRIMARY KEY (COMPNAME));
```

Encapsulation of Operations in SQL

In SQL, a **user-defined type** can have its own behavioral specification by specifying methods (or operations) in addition to the attributes. The general form of a UDT specification with methods is as follows:

```
CREATE TYPE ( < TYPE-NAME> (
      < LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>
     < DECLARATION OF FUNCTIONS (METHODS)>
       );
```

For example, we declared a method Age() that calculates the age of an individual object of type PERSON_TYPE.

```
CREATE TYPE PERSON_TYPE AS ( NAME VARCHAR (35),
              SEX CHAR,
              BIRTH_DATE DATE,
              PHONES PHONE_TYPE ARRAY [4],
             ADDR ADDR_TYPE
  INSTANTIABLE
  NOT FINAL
  REF IS SYSTEM GENERATED
  INSTANCE METHOD AGE() RETURNS INTEGER;
  CREATE INSTANCE METHOD AGE() RETURNS INTEGER
      FOR PERSON_TYPE
      BEGIN
          RETURN /* CODE TO CALCULATE A PERSON’S AGE FROM
          TODAY’S DATE AND SELF.BIRTH_DATE */
     END;
 );
```

In general, a UDT can have a number of user-defined functions associated with it. The syntax is

INSTANCE METHOD <NAME> (<ARGUMENT_LIST>) **RETURNS**;

Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM language of SQL. External functions are written in a host language, with only their signature (interface) appearing in the UDT definition.

An external function definition can be declared as follows:

DECLARE EXTERNAL < FUNCTION_NAM>< SIGNATURE>
LANGUAGE <LANGUAGE_NAME>;

Attributes and functions in UDTs are divided into three categories:

■ PUBLIC (visible at the UDT interface)

■ PRIVATE (not visible at the UDT interface)

■ PROTECTED (visible only to subtypes)

It is also possible to define virtual attributes as part of UDTs, which are computed and updated using functions.

Inheritance and Overloading of Functions in SQL

In SQL, inheritance can be applied to types or to tables; we will discuss the meaning of each in this section. SQL has rules for dealing with **type inheritance** (specified via the UNDER keyword). In general, both attributes and instance methods (operations) are inherited. The phrase **NOT FINAL** must be included in a UDT if subtypes are allowed to be created under that UDT.

Associated with type inheritance are the rules for overloading of function implementations and for resolution of function names. These inheritance rules can be summarized as follows:

■ All attributes are inherited.

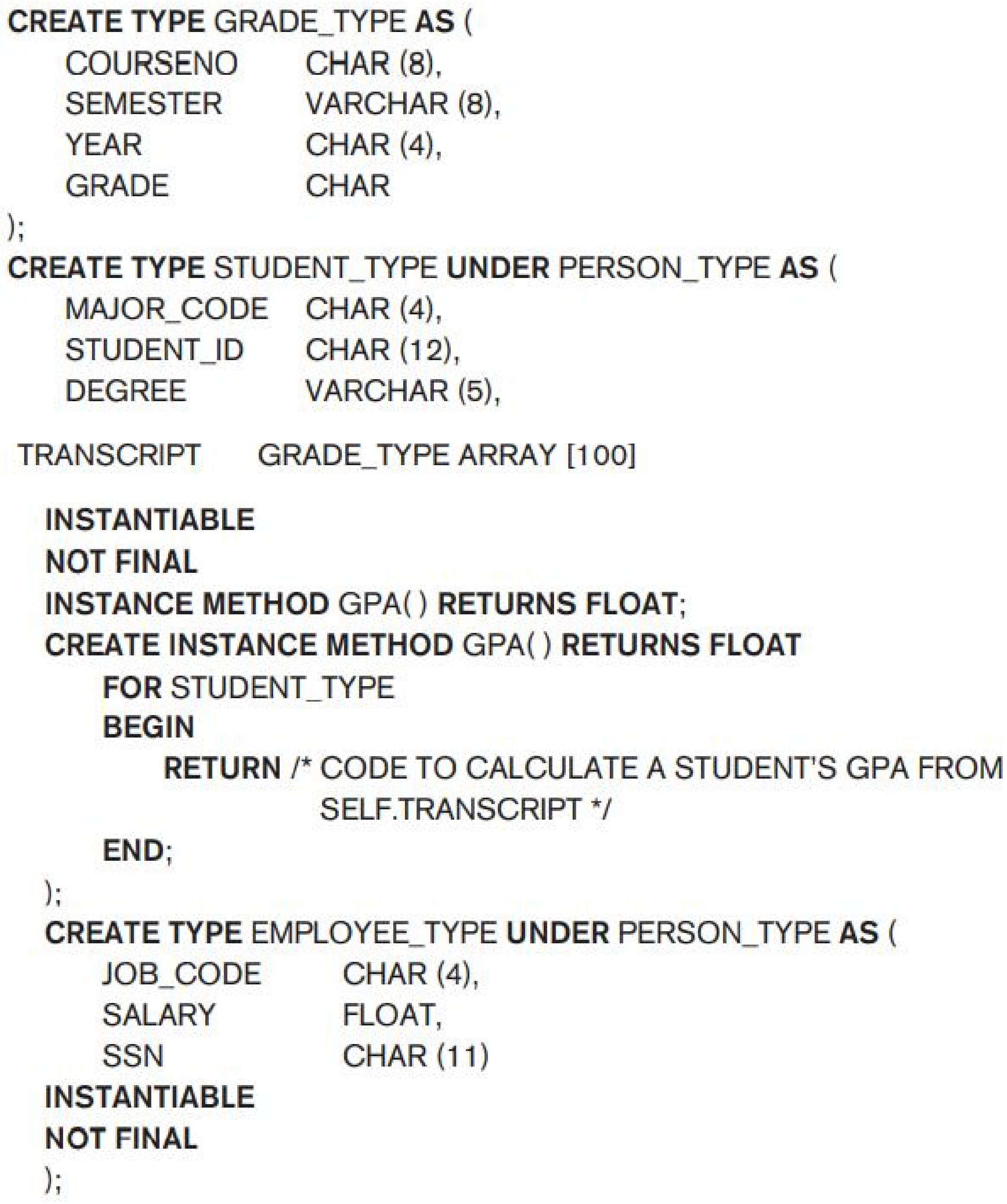
■ The order of supertypes in the UNDER clause determines the inheritance hierarchy. ■ An instance of a subtype can be used in every context in which a supertype instance is used.

■ A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.

■ When a function is called, the best match is selected based on the types of all arguments.

■ For dynamic linking, the types of the parameters are considered at runtime.

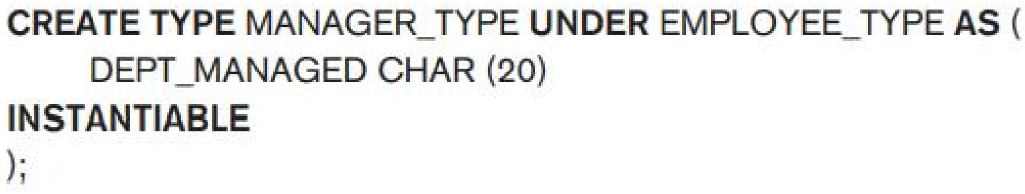
Consider the following examples to illustrate type inheritance, which are illustrated in Figure. Suppose that we want to create two subtypes of PERSON_TYPE: EMPLOYEE_TYPE and STUDENT_TYPE. In addition, we also create a subtype MANAGER_TYPE that inherits all the attributes (and methods) of EMPLOYEE_TYPE but has an additional attribute DEPT_MANAGED. These subtypes are shown in following quary.



```
CREATE TYPE GRADE_TYPE AS (
     COURSENO       CHAR (8),
     SEMESTER       VARCHAR (8),
     YEAR           CHAR (4),
     GRADE          CHAR
);
CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
     MAJOR_CODE     CHAR (4),
     STUDENT_ID     CHAR (12),
     DEGREE         VARCHAR (5),

 TRANSCRIPT      GRADE_TYPE ARRAY [100]

   INSTANTIABLE
   NOT FINAL
   INSTANCE METHOD GPA( ) RETURNS FLOAT;
   CREATE INSTANCE METHOD GPA( ) RETURNS FLOAT
        FOR STUDENT_TYPE
        BEGIN
             RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                          SELF.TRANSCRIPT */
        END;
   );
   CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
        JOB_CODE       CHAR (4),
        SALARY         FLOAT,
        SSN            CHAR (11)
   INSTANTIABLE
   NOT FINAL
   );
```



```
CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
       DEPT_MANAGED CHAR (20)
INSTANTIABLE
);
```

Another facility in SQL is **table inheritance** via the supertable/subtable facility. This is also specified using the keyword UNDER. INSERT, DELETE, and UPDATE operations are appropriately propagated.

Distributed Database

Distributed Database Concepts

A **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.

For a database to be called distributed, the following minimum conditions should be satisfied:

■ **Connection of database nodes over a computer network**: There are multiple computers, called sites or nodes. These sites must be connected by an underlying network to transmit data and commands among sites.

■ **Logical interrelation of the connected databases:** It is essential that the information in the various database nodes be logically related.

■ **Possible absence of homogeneity among connected nodes**: It is not necessary that all nodes be identical in terms of data, hardware, and software.

The sites may all be located in physical proximity—say, within the same building or a group of adjacent buildings—and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul or wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines, cables, wireless communication infrastructures, or satellites. It is common to have a combination of various types of networks.

Transparency

The concept of transparency extends the general idea of hiding implementation details from end users. In a DDB scenario, the data and software are distributed over multiple nodes connected by a computer network, so additional types of transparencies are introduced.

The following types of transparencies are possible:

■ **Data organization transparency (also known as distribution or network transparency).** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency.

■ **Replication transparency**. As we show in Figure 23.1, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability.

■ **Fragmentation transparency**. Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation (table) into subrelations that are subsets of the tuples (rows) in the original relation; this is also known as **sharding** in the newer big data and cloud computing systems. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation.

■ Other transparencies include **design transparency** and execution transparency—which refer, respectively, to freedom from knowing how the distributed database is designed and where a transaction executes.

Availability and Reliability

Reliability and availability are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it.

Scalability and Partition Tolerance

Scalability determines the extent to which the system can expand its capacity while continuing to operate without interruption. There are two types of scalability:

1. Horizontal scalability: This refers to expanding the number of nodes in the distributed system. As nodes are added to the system, it should be possible to distribute some of the data and processing loads from existing nodes to the new nodes.

2. Vertical scalability: This refers to expanding the capacity of the individual nodes in the system, such as expanding the storage capacity or the processing power of a node.

Autonomy

Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution.

Design autonomy refers to independence of data model usage and transaction management techniques among nodes.

Communication autonomy determines the extent to which each node can decide on sharing of information with other nodes.

Execution autonomy refers to independence of users to act as they please.

Advantages of Distributed Databases

Some important advantages of DDB are listed below.

1. Improved ease and flexibility of application development. Developing and maintaining applications at geographically distributed sites of an organization is facilitated due to transparency of data distribution and control.

2. Increased availability. This is achieved by the isolation of faults to their site of origin without affecting the other database nodes connected to the network. When the data and DDBMS software are distributed over many sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site has been replicated at another site prior to the failure, then the user will not be affected at all. The ability of the system to survive network partitioning also contributes to high availability.

3. Improved performance. A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases.

4. Easier expansion via scalability. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more nodes is much easier than in centralized (non-distributed) systems.

Data Fragmentation, Replication, and Allocation Techniques

The techniques that are used to break up the database into logical units, called **fragments,** which may be assigned for storage at the various nodes. **Data replication,** which permits certain data to be stored in more than one site to increase availability and reliability. and the process of **allocating** fragments or replicas of fragments—for storage at the various nodes. These techniques are used during the process of **distributed database design**.

Data Fragmentation and Sharding

In a DDB, decisions must be made regarding which site should be used to store which portions of the database. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites.

Before we decide on how to distribute the data, we must determine the **logical units** of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each whole relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT.

We may want to store the database information relating to each department at the computer site for that department. A technique called **horizontal fragmentation** or **sharding** can be used to partition each relation by department.

Horizontal Fragmentation (Sharding).

A **horizontal fragment** or **shard** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment can be specified by a condition on one or more attributes of the relation, or by some other mechanism.

For example, we may define three horizontal fragments on the EMPLOYEE relation in with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1)—each fragment contains the EMPLOYEE tuples working for a particular department.

Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (Dnum = 5), (Dnum = 4), and (Dnum = 1)—each fragment contains the PROJECT tuples controlled by a particular department.

Horizontal fragmentation divides a relation horizontally by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites (nodes) in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. Thus, related data between the primary and the secondary relations gets fragmented in the same way.

Vertical Fragmentation.

Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation “vertically” by columns. A vertical fragment of a relation keeps only certain attributes of the relation.

For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—Name, Bdate, Address, and Sex—and the second includes work-related information—Ssn, Salary, Super_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately.

It is necessary to include the primary key or some unique key attribute in every vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.

Mixed (Hybrid) Fragmentation.

We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments.

In this case, the original relation can be reconstructed by applying UNION and OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a fragment of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi L(\sigma C(R))$.

A **fragmentation schema** of a database is a definition of a set of fragments that includes all attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations.

An **allocation schema** describes the allocation of fragments to nodes (sites) of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored.

Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the whole database at every site in the distributed system, thus creating a **fully replicated distributed database**. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval (read performance) for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module.

The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case, all fragments must be disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.

Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called data distribution (or data allocation).

Example of Fragmentation, Allocation, and Replication

Suppose that the company has three computer sites— one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees who work in that department and the projects controlled by that department. Further, we assume that these sites mainly access the Name, Ssn, Salary, and Super_ssn attributes of EMPLOYEE. Site 1 is used by company headquarters and accesses all employee and project information regularly.

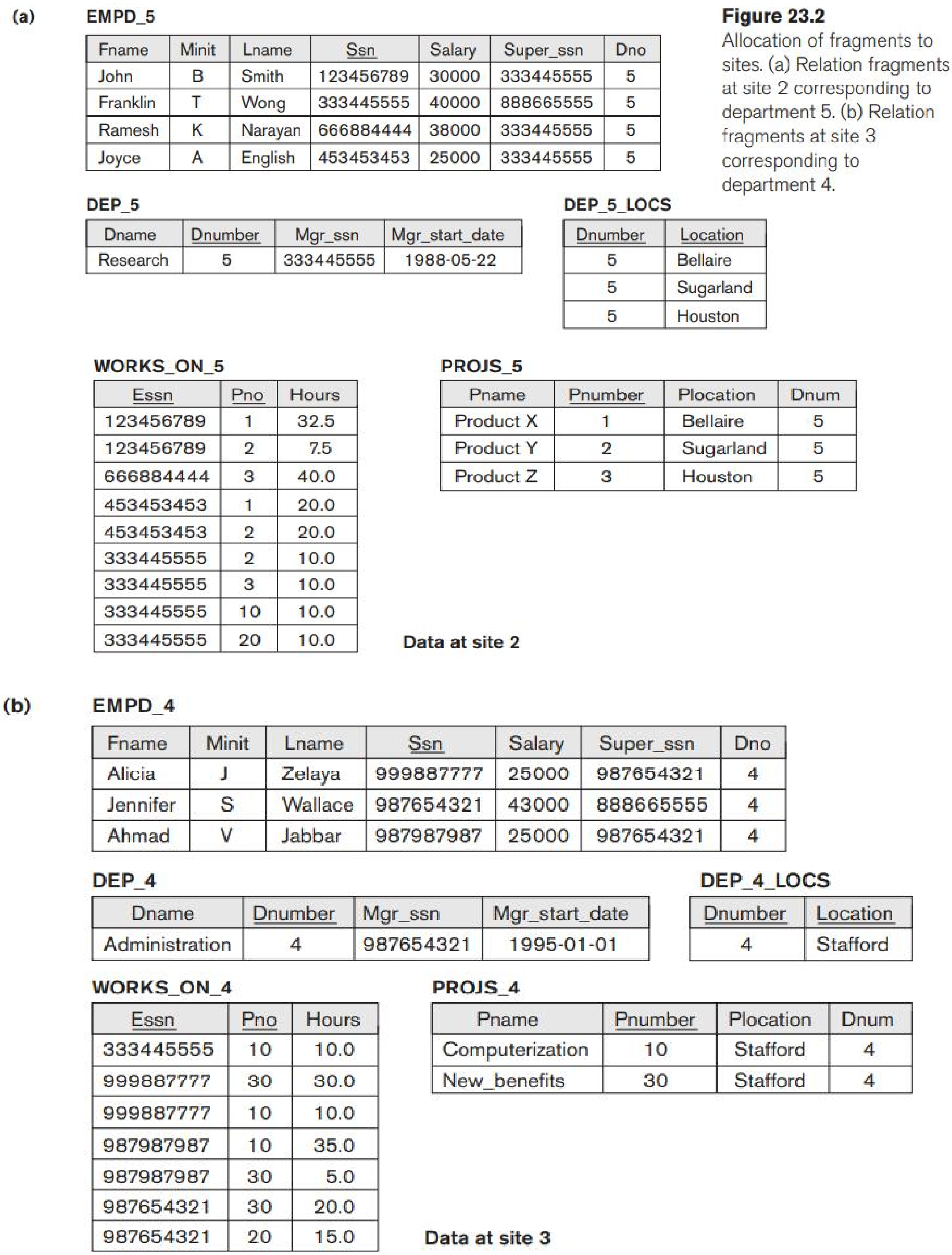


Figure 23.2
Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

(a)

EMPD_5

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

DEP_5

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

DEP_5_LOCS

Dnumber	Location
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON_5

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

PROJS_5

Pname	Pnumber	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

Data at site 2

(b)

EMPD_4

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP_4

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

DEP_4_LOCS

Dnumber	Location
4	Stafford

WORKS_ON_4

Essn	Pno	Hours
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0

PROJS_4

Pname	Pnumber	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

Data at site 3

Types of Distributed Database Systems

The term **distributed database management system** can describe various systems that differ from one another in many respects

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous;** otherwise, it is called **heterogeneous.**

Another factor related to the degree of homogeneity is the **degree of local autonomy**. If there is no provision for the local site to function as a standalone DBMS, then the system has **no local autonomy**.

Figure: Classification of distributed databases.

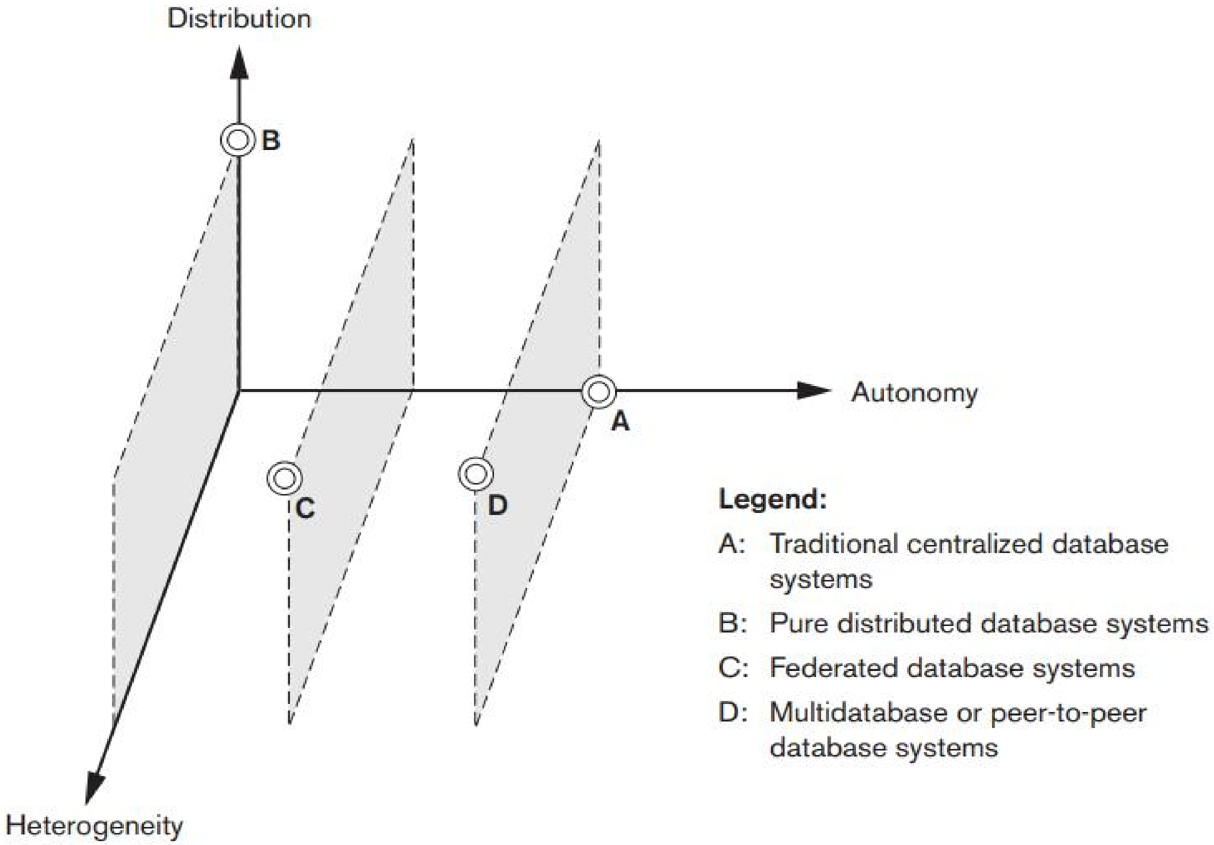

Distribution
B
Autonomy
A
C
D
Heterogeneity
Legend:
A: Traditional centralized database systems
B: Pure distributed database systems
C: Federated database systems
D: Multidatabase or peer-to-peer database systems

Figure shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is complete autonomy but a total lack of distribution and heterogeneity (point A in the figure).

At one extreme of the autonomy spectrum, we have a DDBMS that looks like a centralized DBMS to the user, with zero autonomy (point B).

Along the autonomy axis we encounter two types of DDBMSs called federated database system (point C) and multidatabase system (point D).

The term **federated database system (FDBS**) is used when there is some global view or schema of the federation of databases that is shared by the applications (point C). On the other hand, a **multidatabase system** has full local autonomy in that it does not have a global schema but interactively constructs one as needed by the application (point D). Both systems are hybrids between distributed and centralized systems.

Federated Database Management Systems Issues

The type of heterogeneity present in FDBSs may arise from several sources.

1. **Differences in data models:** Databases in an organization come from a variety of data models, including the so-called legacy models (hierarchical and network), the relational data model, the object data model, and even files..
2. **Differences in constraints:** Constraint facilities for specification and implementation vary from system to system
3. **Differences in query languages**: Even with the same data model, the languages and their versions vary.

Semantic Heterogeneity: Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases.

Query Processing in Distributed Databases

Distributed Query Processing

A distributed database query is processed in stages as follows:

1. Query Mapping. The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performed in a centralized DBMS.

2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.

3. Global Query Optimization. Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).

4. Local Query Optimization. This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

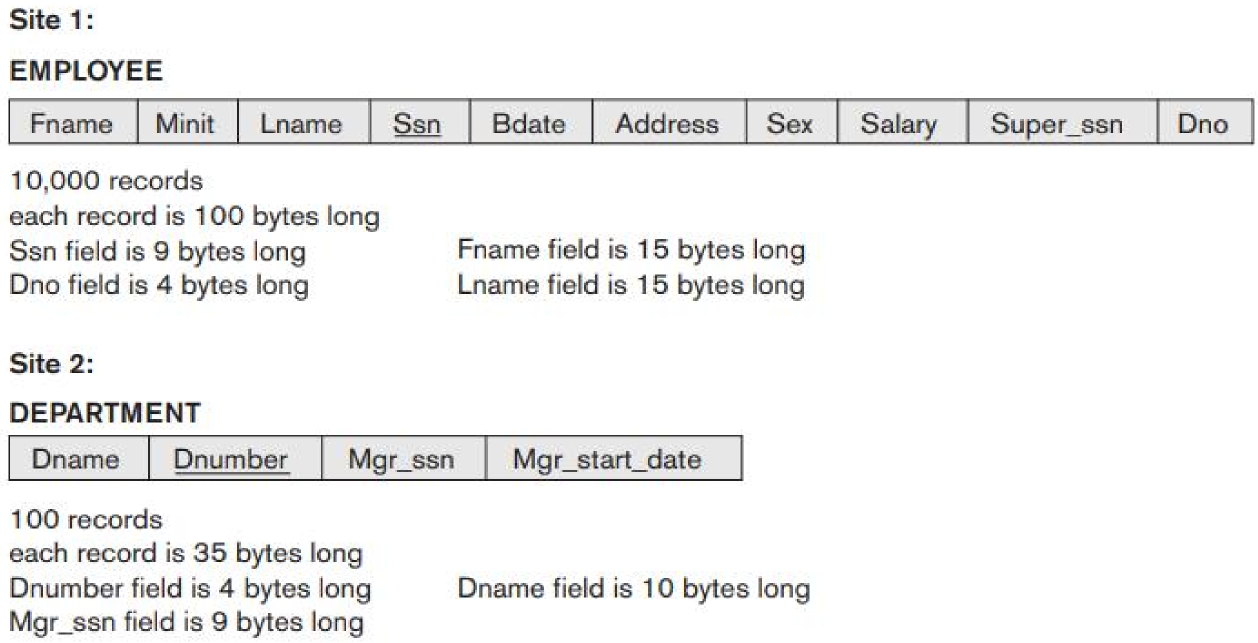
Data Transfer Costs of Distributed Query Processing

In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed.

Hence, DDBMS query optimization algorithms consider the goal of reducing the amount of data transfer as an optimization criterion in choosing a distributed query execution strategy.

Suppose that the EMPLOYEE and DEPARTMENT relations are distributed at two sites as shown in Figure. We will assume in this example that neither relation is fragmented. According to Figure, the size of the EMPLOYEE relation is 100 * 10,000 = 106 bytes, and the size of the DEPARTMENT relation is 35 * 100 = 3,500 bytes.

Figure: Example to illustrate volume of data transferred.



Site 1:

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

10,000 records
each record is 100 bytes long

Ssn field is 9 bytes long	Fname field is 15 bytes long
Dno field is 4 bytes long	Lname field is 15 bytes long

Site 2:

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

100 records
each record is 35 bytes long

Dnumber field is 4 bytes long	Dname field is 10 bytes long
Mgr_ssn field is 9 bytes long	

Consider the query Q: For each employee, retrieve the employee name and the name of the department for which the employee works. This can be stated as follows in the relational algebra:

$$Q: \pi_{\text{Fname,Lname,Dname}}(\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT})$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is 40 bytes long. The query is submitted at a distinct site 3, which is called the result site because the query result is needed there.

There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred

2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.

3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 400,000 + 3,500 = 403,500 bytes must be transferred.

Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the **semijoin operation** is to reduce the number of tuples in a relation before transferring it to another site.

Intuitively, the idea is to send the joining column of one relation R to the site where the other relation S is located; this column is then joined with S. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with R. Hence, only the joining column of R is transferred in one direction, and a subset of S with no extraneous tuples or attributes is transferred in the other direction.

The semijoin operation was devised to formalize this strategy. A semijoin operation $R \ltimes_{A=B} S,$ where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression

πR($R \ltimes_{A=B} S,$). In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring F = πB(S) to the site where R resides and then joining F with R, thus leading to the strategy discussed here. Notice that the semijoin operation is not commutative; that is,

$$R \ltimes S \neq S \ltimes R$$

Query and Update Decomposition

queries, a **query decomposition** module must break up or decompose a query into subqueries that can be executed at the individual sites. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment.

For example, consider the query Q: ***Retrieve the names and hours per week for each employee who works on some project controlled by department 5***.

This can be specified in SQL as follows:

Q: **SELECT** Fname, Lname, Hours

FROM EMPLOYEE, PROJECT, WORKS_ON

WHERE Dnum=5 AND Pnumber=Pno AND Essn=Ssn;

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS_ON5 that all tuples satisfying the conditions (Dnum = 5 AND Pnumber = Pno) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

$$T_1 \leftarrow \pi_{\text{Essn}}(\text{PROJS5} \bowtie_{\text{Pnumber=Pno}} \text{WORKS\_ON5})$$

$$T_2 \leftarrow \pi_{\text{Essn, Fname, Lname}}(T_1 \bowtie_{\text{Essn=Ssn}} \text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Hours}}(T_2 * \text{WORKS\_ON5})$$

Concurrency Control and Recovery in Distributed Databases

For concurrency control and recovery purposes, numerous problems arise in a distributed DBMS environment that are not encountered in a centralized DBMS environment. These include the following:

■ **Dealing with multiple copies of the data items:** The concurrency control method is responsible for maintaining consistency among these copies. The recovery method is responsible for making a copy consistent with other copies if the site on which the copy is stored fails and recovers later.

■ **Failure of individual sites:** The DDBMS should continue to operate with its running sites, if possible, when one or more individual sites fail. When a site recovers, its local database must be brought up-to-date with the rest of the sites before it rejoins the system.

■ **Failure of communication links**: The system must be able to deal with the failure of one or more of the communication links that connect the sites. An extreme case of this problem is that network partitioning may occur. This breaks up the sites into two or more partitions, where the sites within each partition can communicate only with one another and not with sites in other partitions.

■ **Distributed commit**: Problems can arise with committing a transaction that is accessing databases stored on multiple sites if some sites fail during the commit process. The two-phase commit protocol (see Section 21.6) is often used to deal with this problem.

■ **Distributed deadlock**: Deadlock may occur among several sites, so techniques for dealing with deadlocks must be extended to take this into account.

Distributed Concurrency Control Based on a Distinguished Copy of a Data Item

To deal with replicated data items in a distributed database, a number of concurrency control methods have been proposed that extend the concurrency control techniques that are used in centralized databases.

Similar extensions apply to other concurrency control techniques. The idea is to designate a particular copy of each data item as a **distinguished copy**. The locks for this data item are associated with the distinguished copy.

A number of different methods are based on this idea, but they differ in their method of choosing the distinguished copies. In the **primary site technique**, all distinguished copies are kept at the same site. A modification of this approach is the primary site with a **backup site**. Another approach is the **primary copy** method, where the distinguished copies of the various data items can be stored in different sites. A site that includes a distinguished copy of a data item basically acts as the **coordinator site** for concurrency control on that item.

Primary Site Technique. In this method, a single primary site is designated to be the coordinator site for all database items. Hence, all locks are kept at that site, and all requests for locking or unlocking are sent there.

Primary Site with Backup Site. This approach addresses the second disadvantage of the primary site method by designating a second site to be a **backup site**

Primary Copy Technique. This method attempts to distribute the load of lock coordination among various sites by having the distinguished copies of different data items stored at different sites.

Distributed Concurrency Control Based on Voting

In the **voting method**, there is no distinguished copy; rather, a lock request is sent to all sites that includes a copy of the data item. Each copy maintains its own lock and can grant or deny the request for it. If a transaction that requests a lock is granted that lock by a majority of the copies, it holds the lock and informs all copies that it has been granted the lock. If a transaction does not receive a majority of votes granting it a lock within a certain time-out period, it cancels its request and informs all sites of the cancellation.

The voting method is considered a truly distributed concurrency control method, since the responsibility for a decision resides with all the sites involved.

Distributed Recovery

The recovery process in distributed databases is quite involved. In some cases it is difficult even to determine whether a site is down without exchanging numerous messages with other sites. For example, suppose that site X sends a message to site Y and expects a response from Y but does not receive it.

There are several possible explanations:

■ The message was not delivered to Y because of communication failure.

■ Site Y is down and could not respond.

■ Site Y is running and sent a response, but the response was not delivered.

Another problem with distributed recovery is **distributed commit**. When a transaction is updating data at several sites, it cannot commit until it is sure that the effect of the transaction on every site cannot be lost. This means that every site must first have recorded the local effects of the transactions permanently in the local site log on disk. The two-phase commit protocol is often used to ensure the correctness of distributed commit.

****