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

MCA DEPARTMENT

DATABASE MANAGEMENT SYSTEMS

I MCA

Presented by

V.SARALA

M.C.A DEPARTMENT

MCA-20203 DATABASE MANAGEMENT SYSTEMS

Instruction: 4Periods/week	**Time: 3 Hours**	**Credits:4**
Internal: 25Marks	**External:75Marks**	**Total: 100Marks**

UNIT I

Database and Database Users: Data models, schemas, and instances, three-schemas architecture and data independence, database languages and interfaces, the database system environment, Centralized and client/server architectures for DBMSs, Classification of database management system.

Data Modeling Using the Entity-Relationship Model: Using High—Level Conceptual data model, Entity types, entity sets Attributes and keys, Relationships types, relationship sets, roles and structural constraints, Weak Entity types, ER diagrams Meaning conventions and design issues, Enhance Entity Relationship model,

Relational data model and relational database constraints: Relational model constraints and relational schemas, update operations.

UNIT II

Relational Algebra and Relational Calculus: Unary Relational operations, Relational Algebra operations, Binary Relational operation, Additional Relational operation, Examples of Queries in Relational Algebra, Domain Relational Calculus.

Relational database design by ER and EER Relational Mapping: Relational database design using ER to Relational Mapping, Mapping EER Model Construct to Relations, **Schema Definition, Basic Constraints and Queries:** SQL Data definition, Specifying basic constraints in SQL, Schema change Statements in SQL, Basic queries in SQL, More complex SQL queries, INSERT DELETE UPDATE queries in SQL, Views in SQL, Data base stored Procedures.

UNIT III

Relational Database Design: Informal design Guide lines for Relation Schema, Functional Dependences, Normal forms based on Primary keys, General definitions of Second and Third Normal form, BOYCE-CODE Normalform, Algorithm for Relational database schemadesign, Multi-valued dependencies and fourth Normal forms,

File Organization and Indexes: Introduction, Secondary Storage Devices, Buffering Blocks, placing file records on disk, Operations on Files, Hashing Techniques, Parallelizing Disk Access using RAID Technology, Indexing Structures for files.

UNIT IV

Algorithm for query processing and Optimization: Translating SQL Queries into Relational Algebra, Algorithms for SELECT and JOIN Operations, Algorithms for PROJECT and SET Operations,

Introduction to Transaction Processing Concepts and Theory: Introduction to Transaction Process, Transaction and System Concepts, Characterizing Schedules, Concurrency Control Techniques, Database Recovery Concepts, Recovery Techniques.

Text Book:

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

References Books:

1.Database Management Systems, Raghu Ramakrishnan, Johannes Gehrke, McGraw- Hill.

2.Database Concepts, Abraham Silberschatz, Henry F Korth, S Sudarshan, McGraw-Hill

DATABASE MANAGEMENT SYSTEMS

UNIT - I

Database and Database Users

Introduction

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**.

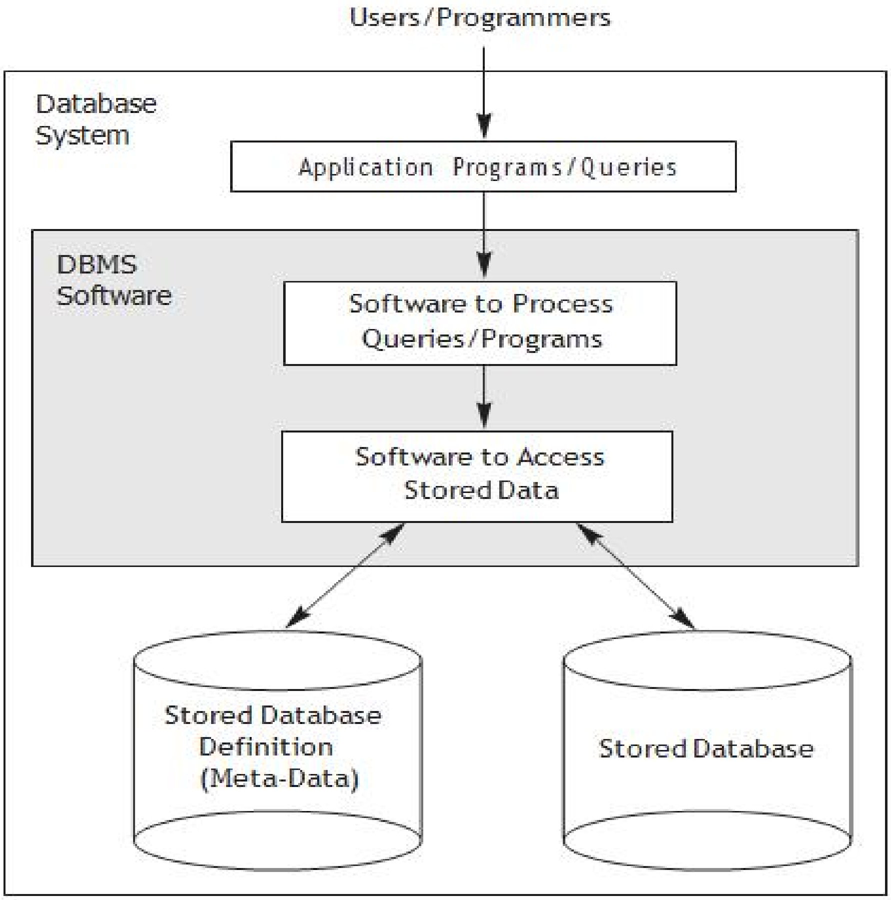

Users/Programmers
Database System
Application Programs/Queries
DBMS Software
Software to Process Queries/Programs
Software to Access Stored Data
Stored Database Definition (Meta-Data)
Stored Database

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.

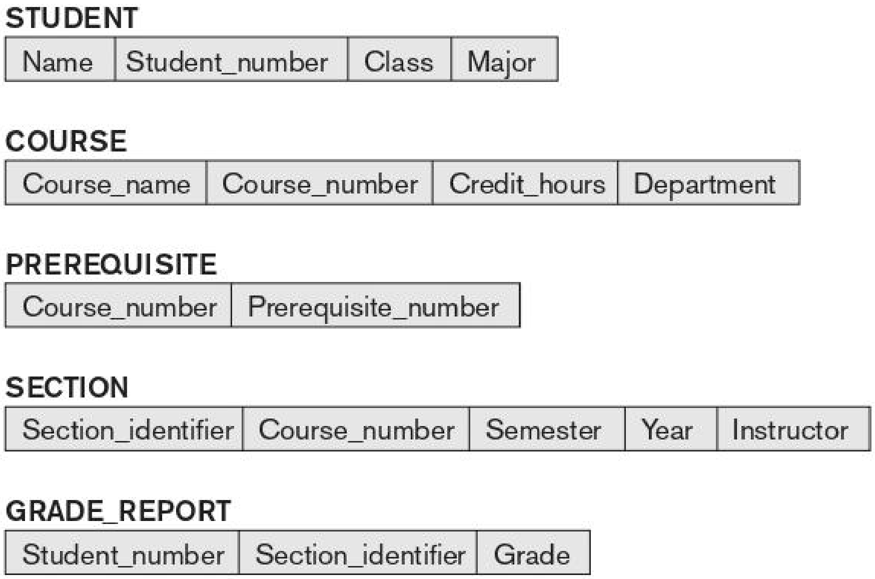
'**Object data model'** as a new family of higher-level implementation data models that are closer to conceptual data models.

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.

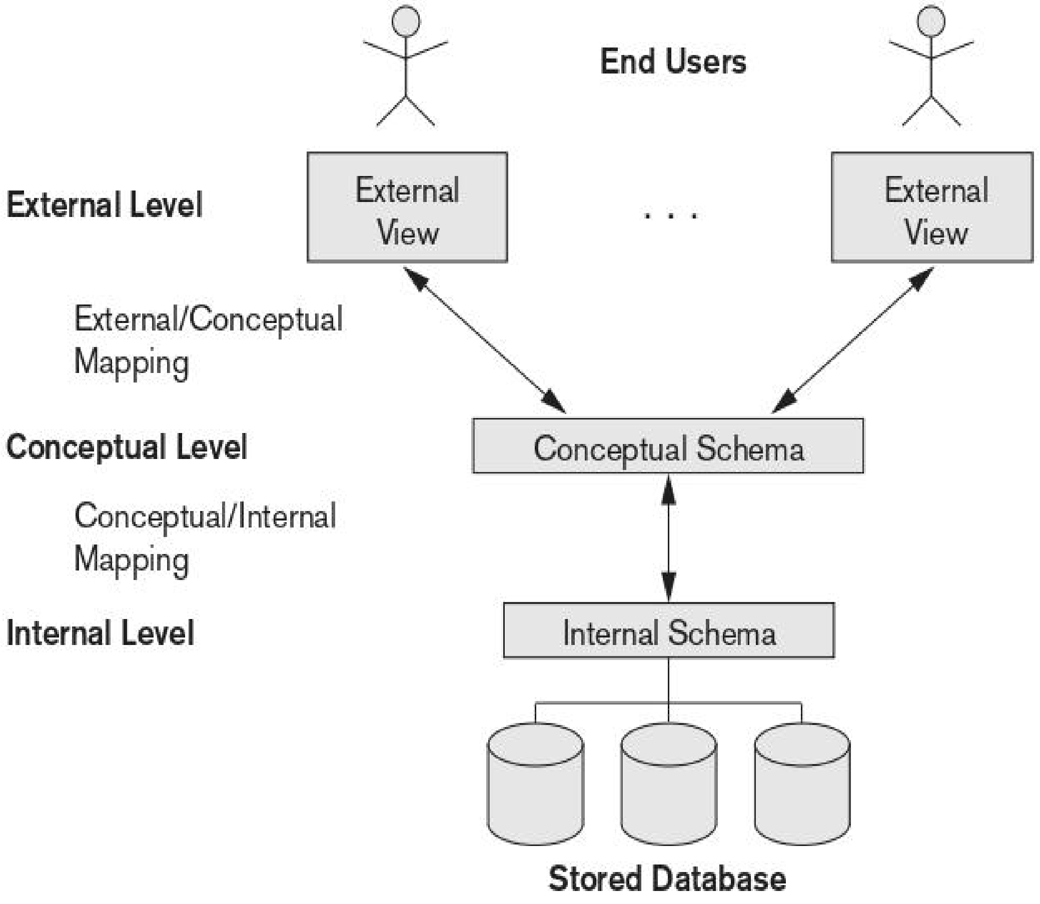
The data in the database at a particular moment in time is called a **database state** or **snapshot**. It is also called the current set of **occurrences** or **instances** in the database. In a given database state, each schema construct has its own current set of instances. For example, the STUDENT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

Database Languages and Interfaces

DBMS Languages

In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language** (DDL), is used by the DBA and by database designers to define both schemas. DDL is used to specify the conceptual schema only.

Storage definition language (SDL): is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages.

View definition language (VDL): to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Data manipulation language (DML): Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language** (DML) for these purposes.

There are two main types of DMLs.

1. **High-level or nonprocedural DML** can be used on its own to specify complex database operations concisely.
2. **Lowlevel or procedural DML** must be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects. Low-level DMLs are also called record-at-a-time DMLs.

Whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**.

A high-level DML used in a standalone interactive manner is called a **query language.**

DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

Menu-based Interfaces for Web Clients or Browsing: These interfaces present the user with lists of options called **menus,** that lead the user through the formulation of a request. Pull-down menus are a very popular technique in Web-based user interfaces. They are also often used in **browsing interfaces**.

Apps for Mobile Devices: These interfaces present mobile users with access to their data. For example, banking, reservations, and insurance companies, among many others, provide apps that allow users to access their data through a mobile phone or mobile device. The apps have built-in programmed interfaces that typically allow users to login using their account name and password; the apps then provide a limited menu of options for mobile access to the user data, as well as options such as paying bills (for banks) or making reservations (for reservation Web sites).

Forms-based Interfaces: A forms-based interface displays a 'form' to each user. Users can fill out all of the form entries to insert new data. Many DBMSs have 'forms specification languages', which are special languages that help programmers specify such forms.

Graphical User Interfaces: A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms.

Natural Language Interfaces: These interfaces accept requests written in English or some other language and attempt to understand them. A natural language interface usually has its own schema, which is similar to the database conceptual schema, as well as a dictionary of important words.

Keyword-based Database Search: These are somewhat similar to Web search engines, which accept strings of natural language (like English or Spanish) words and match them with documents at specific sites (for local search engines) or Web pages on the Web at large (for engines like Google or Ask).

Speech Input and Output: Limited use of speech as an input query and speech as an answer to a question or result of a request is becoming commonplace. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

Interfaces for Parametric Users: Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface.

Interfaces for the DBA: Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

The Database System Environment

1. DBMS Component Modules

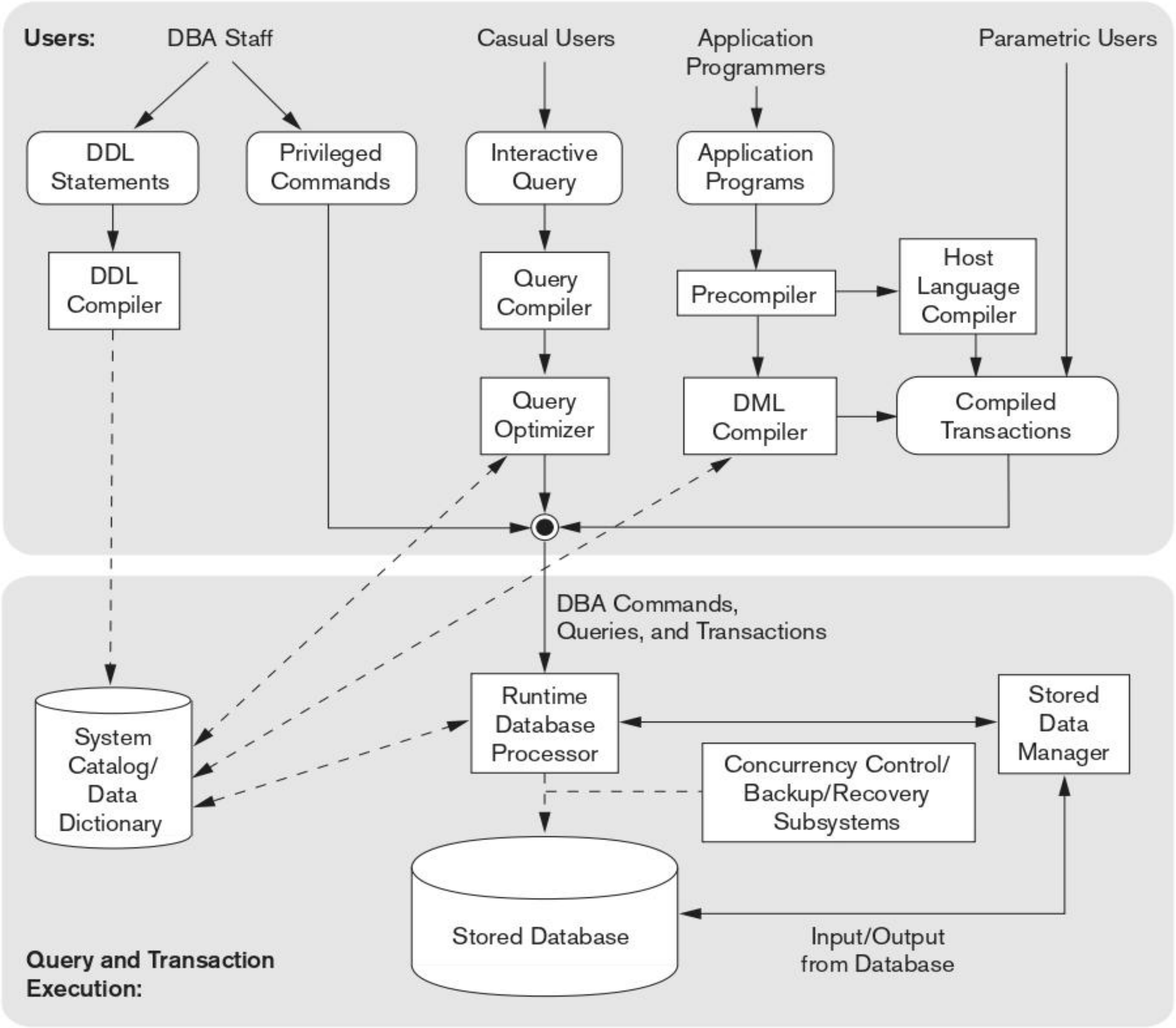


Figure 2.3
Component modules of a DBMS and their interactions.

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

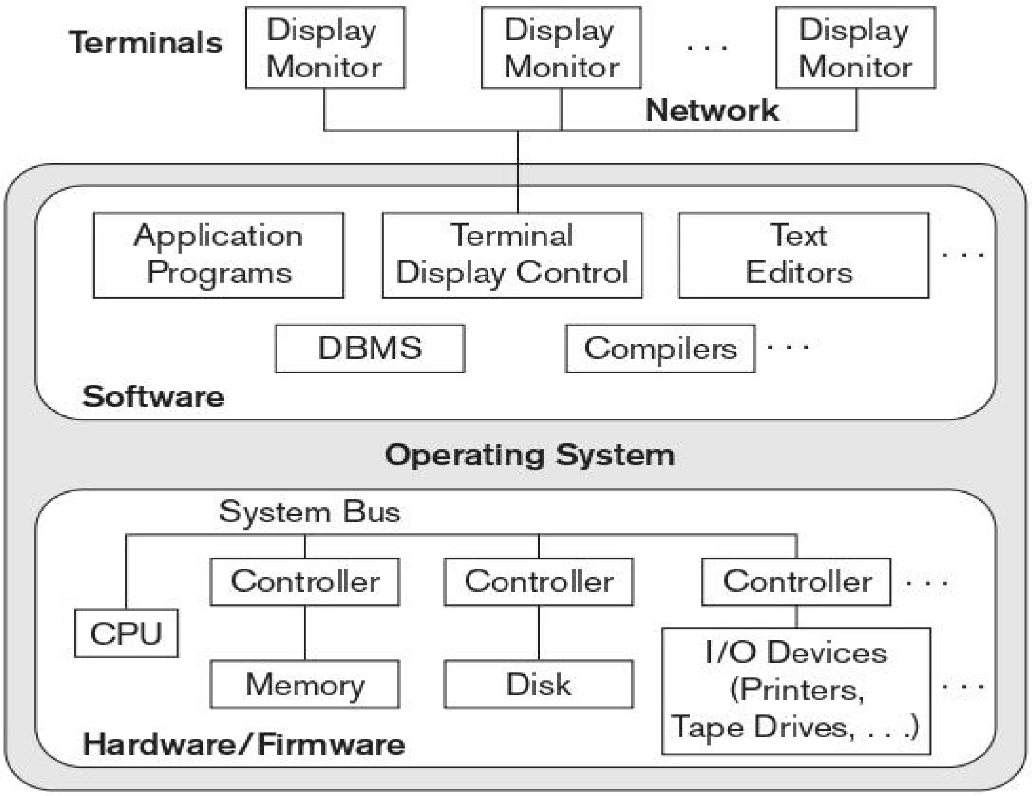
1. Centralized DBMSs Architecture:

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.

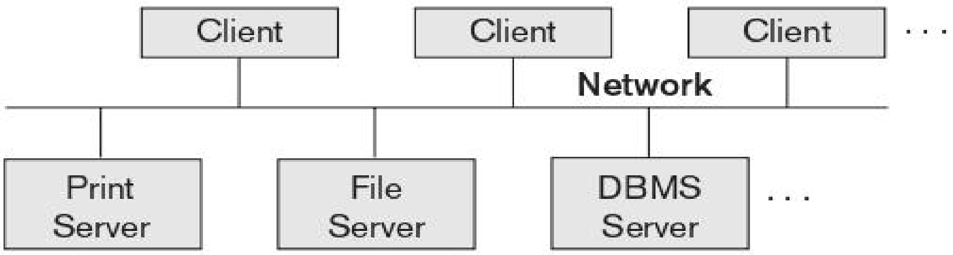
Figure: A Physical Centralized architecture



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



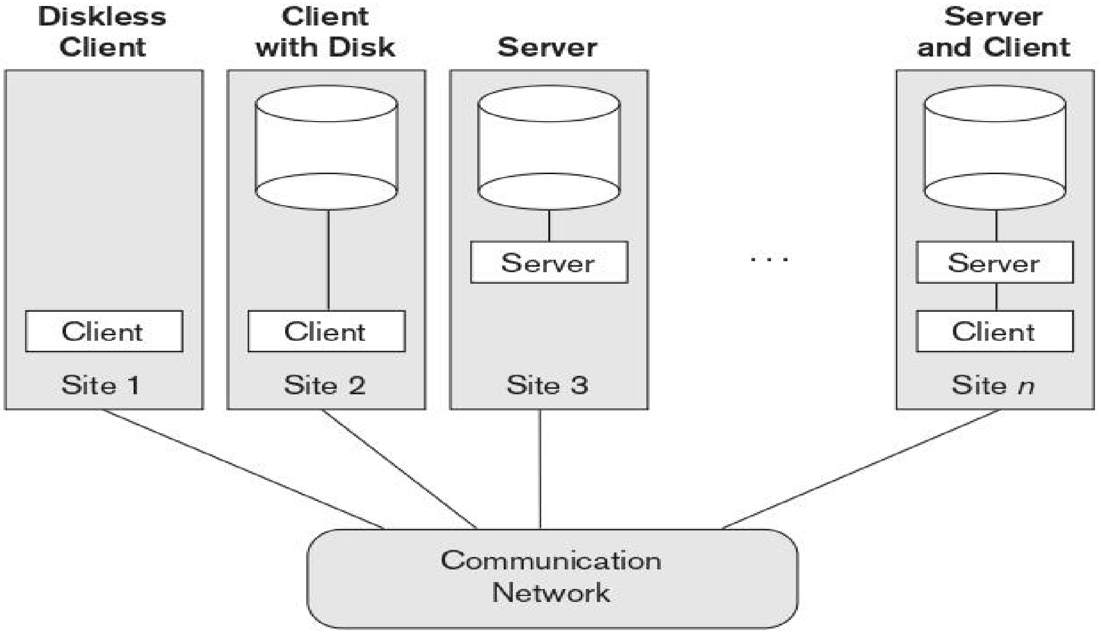
A **file server** that maintains the files of the client machines. A **printer server** connected to various printers; thereafter, all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** are fall into the specialized server category. **Specialized servers**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



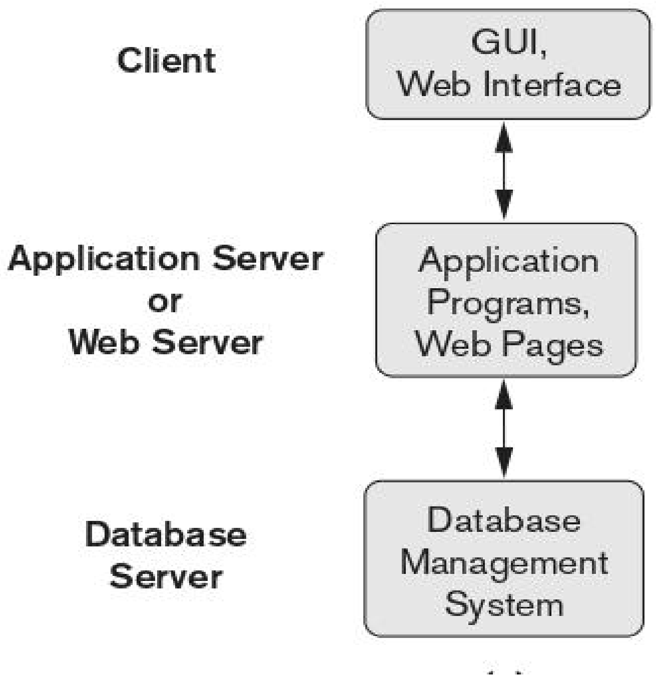
The Client/Server architecture is increasingly being incorporated into commercial DBMS packages. In relational database management systems (RDBMSs), many of which started as centralized systems. Because SQL provided a standard language for RDBMSs, this created a logical dividing point between client 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



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.

Classification of Database Management Systems

Several criteria can be used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**.

The **object data model** was implemented in some commercial systems but has not had widespread use.

Many legacy (older) applications still run on database systems based on the **hierarchical and network data models**.

The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has a new class of DBMSs called **object-relational DBMSs**.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with Personal Computers. **Multiuser systems**, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users. A **distributed DBMS (DDBMS)** can have the actual database and DBMS software distributed over many sites connected by a computer network. **Homogeneous** DDBMSs use the same DBMS software at all the sites.

The fourth criterion is **cost of the DBMS.** The majority of DBMS packages cost between $10,000 and $100,000. Single-user low-end systems that work with microcomputers cost between $100 and $3000. A few elaborate packages cost more than $100,1000.

We can also classify a DBMS on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures.

Finally, a DBMS can be **general purpose** or **special purpose.**

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:

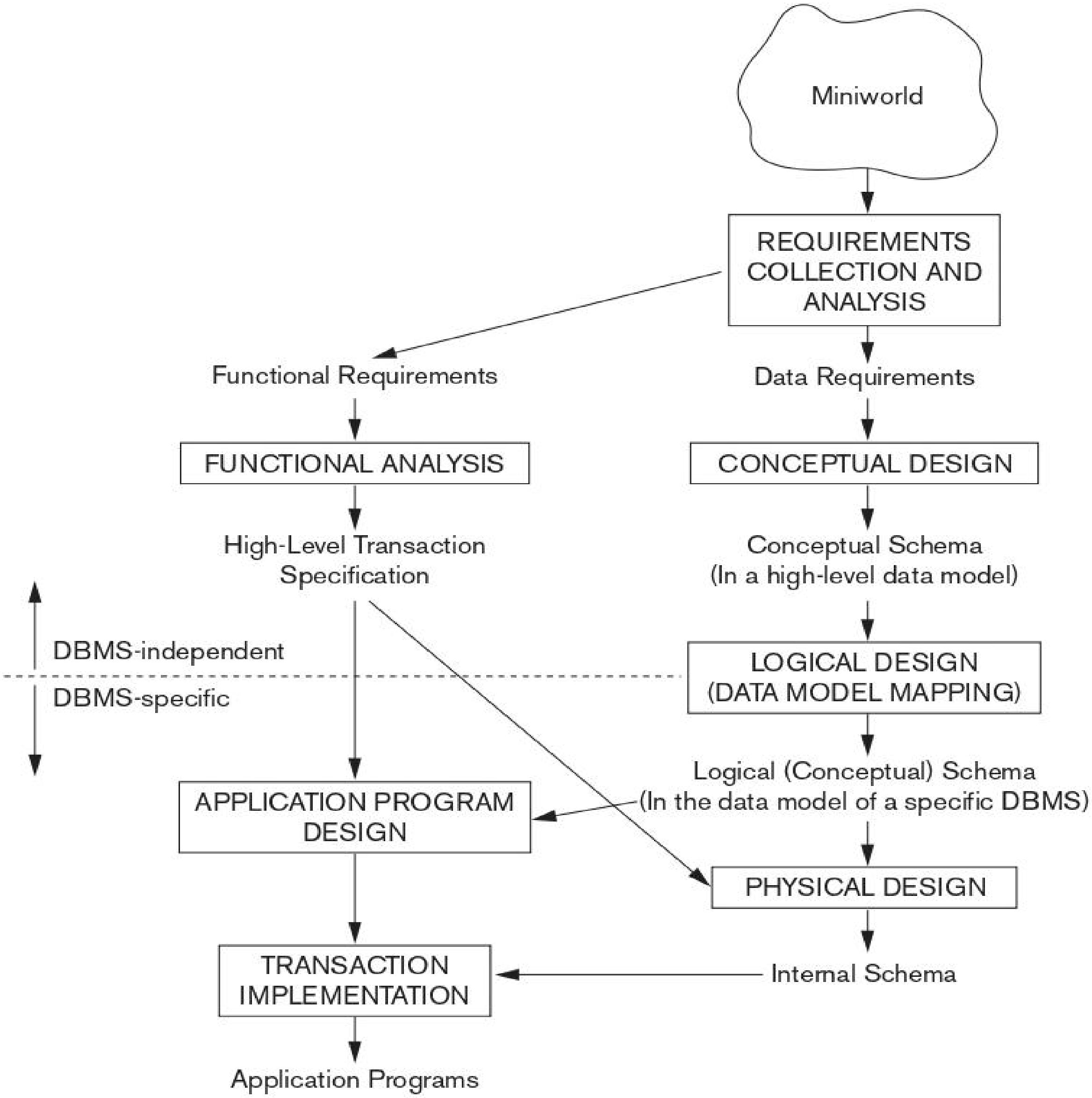

Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Data Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (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

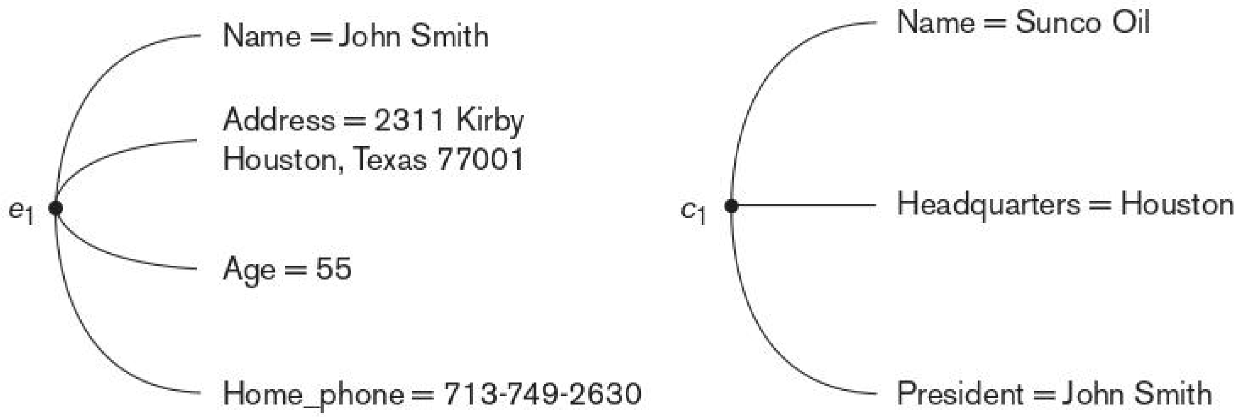
The ER model describes data as entities, relationships, and attributes.

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.

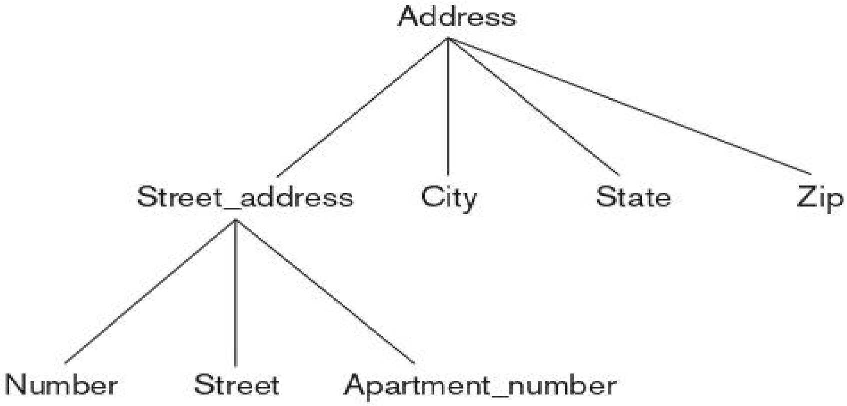
The COMPANY entity c1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively.

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**.

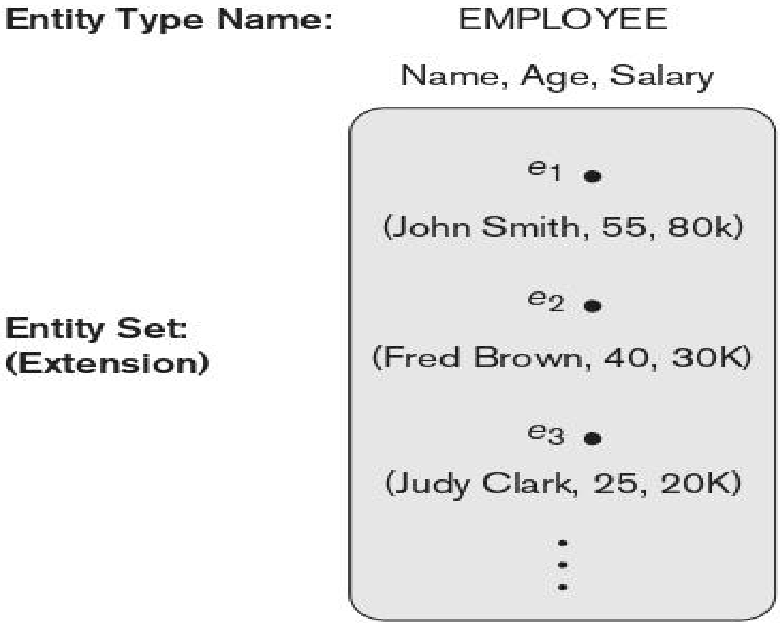
Eg: {Address_phone({Phone(Area_code,Phone_number)},Address(Street_address (Number,Street,Apartment_number),City,State,Zip))}

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:



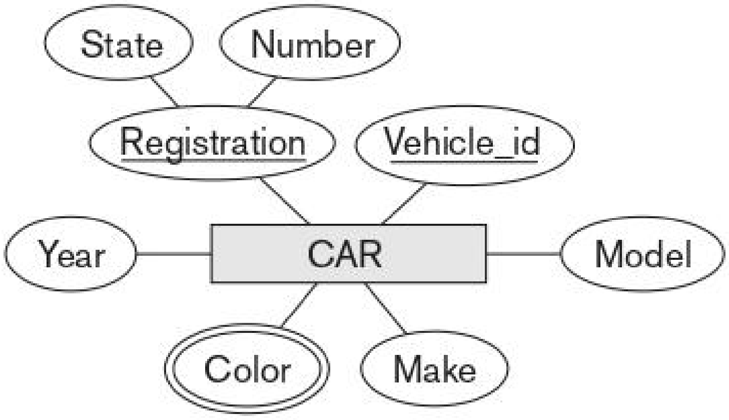
:

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.

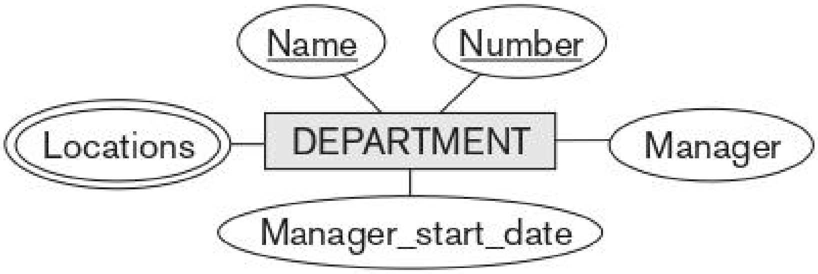


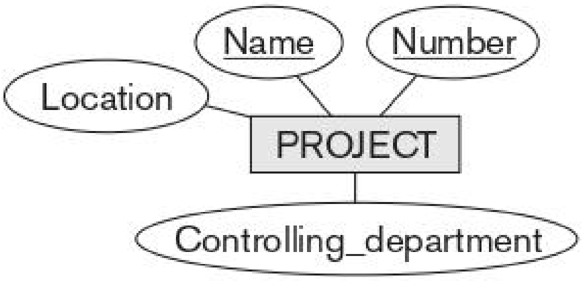
3. Initial Conceptual Design of the COMPANY Database

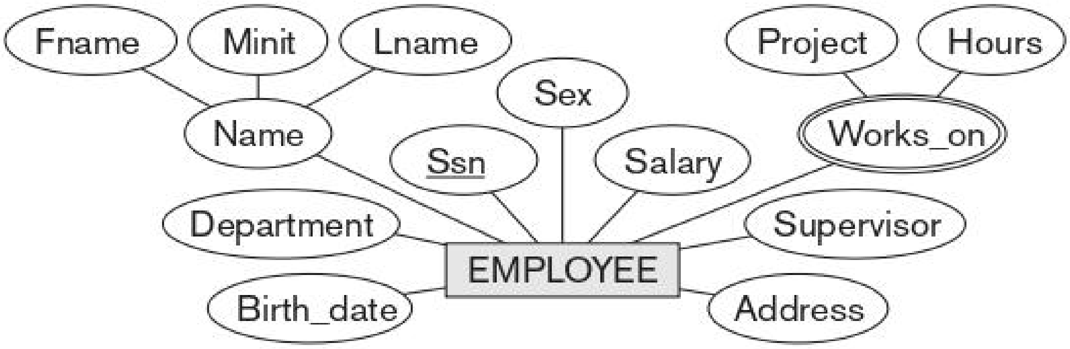
We can define the entity types for the COMPANY database. We can identify four entity types—one corresponding to each of the four items in the specification:

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

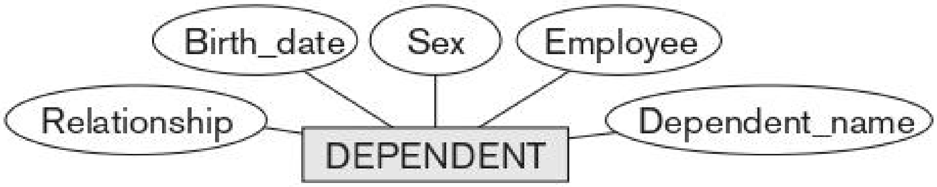
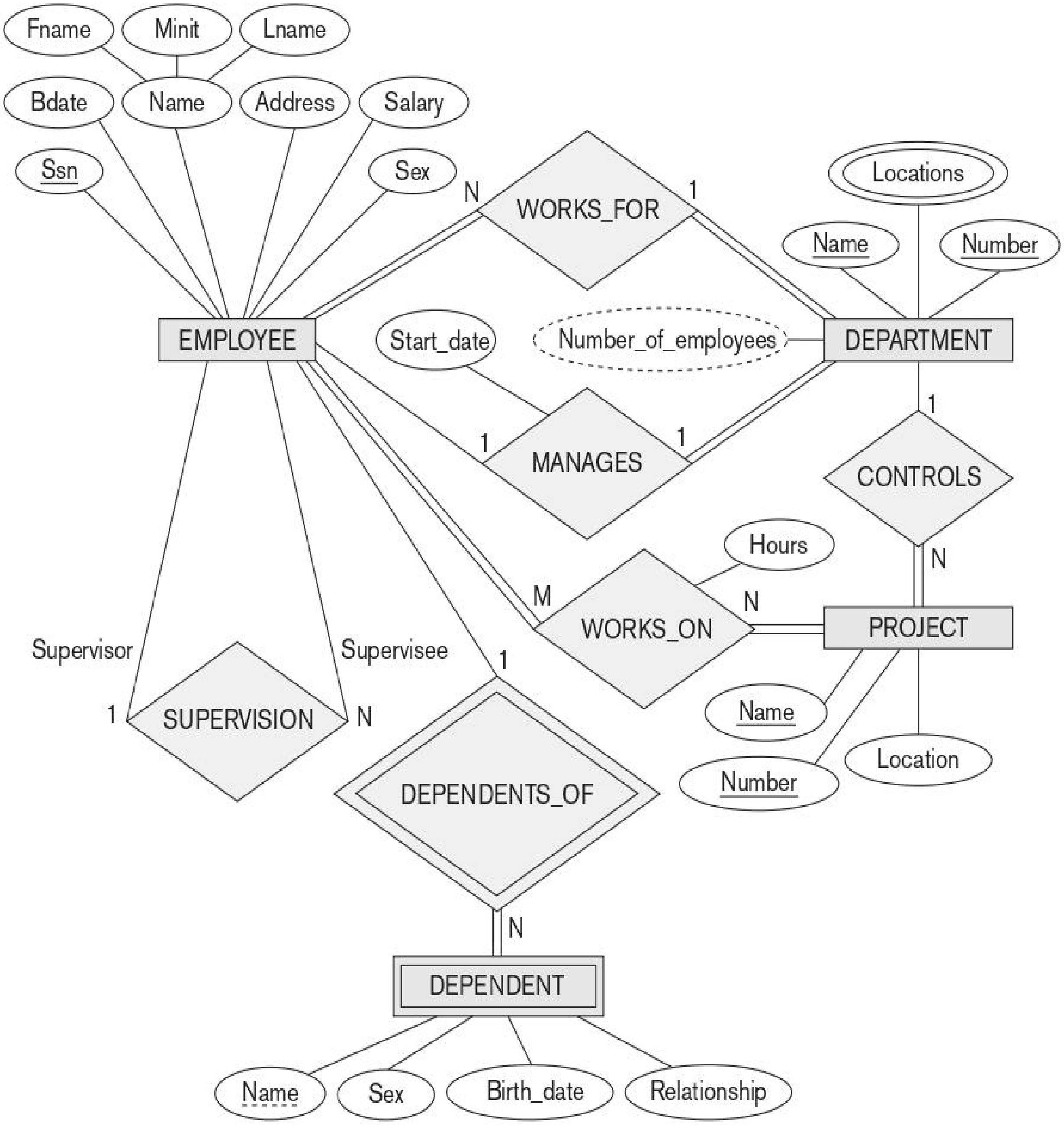

Birth_date
Sex
Employee
Relationship
Dependent_name
DEPENDENT

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



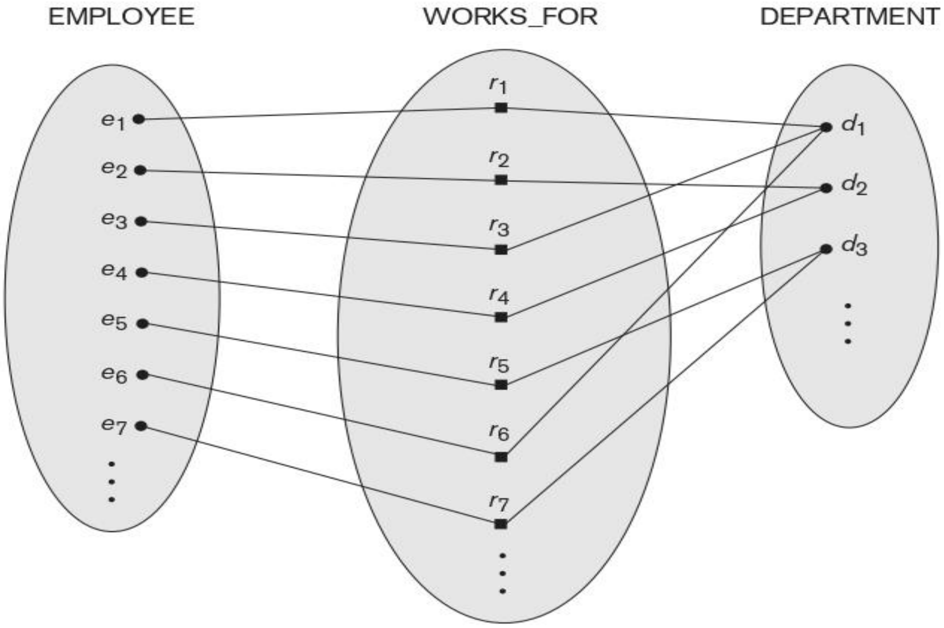
Relationship Types, Relationship Sets, Roles and Structural Constraints

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.

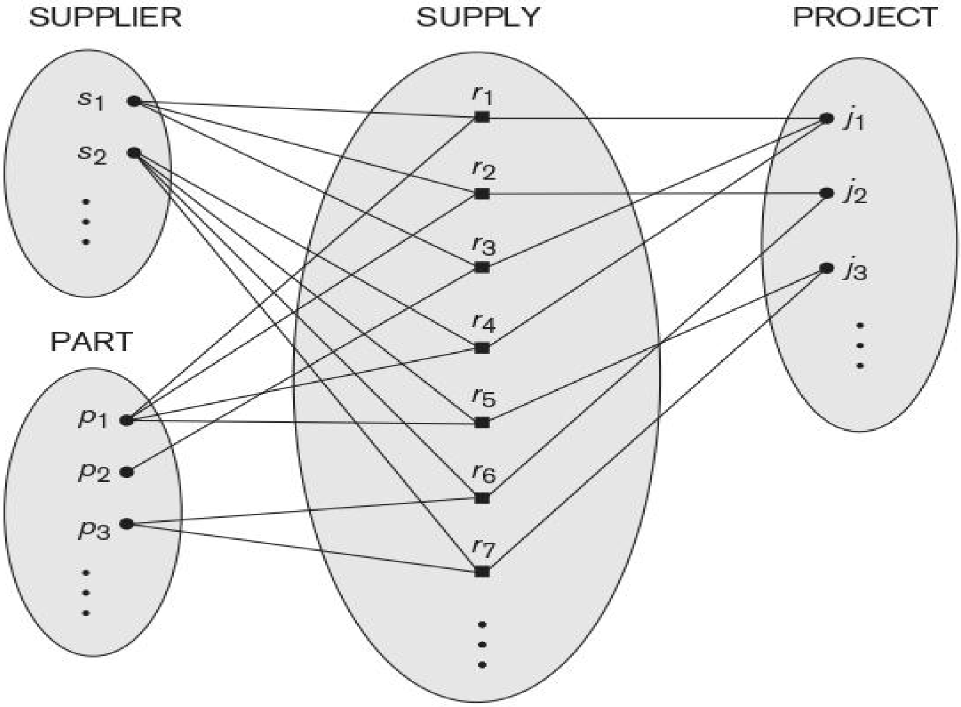
In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shape box.

2. Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type: The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary,** and one of degree three is called **ternary**.

An example of a ternary relationship is SUPPLY, shown in Figure, where each relationship instance **r$_i$** associates three entities, a supplier **s,** a part **p**, and a project **j** whenever **s** supplies part **p** to project **j**.

Figure: Some relationship instances in the SUPPLY ternary relationship set.



Relationships as Attributes:

It is sometimes convenient to think of a binary relationship type in terms of attributes. Consider the WORKS_FOR relationship type. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is (a reference to) the DEPARTMENT entity for which that employee works. Hence, the value set for this Department attribute is the set of all DEPARTMENT entities, which is the DEPARTMENT entity set.

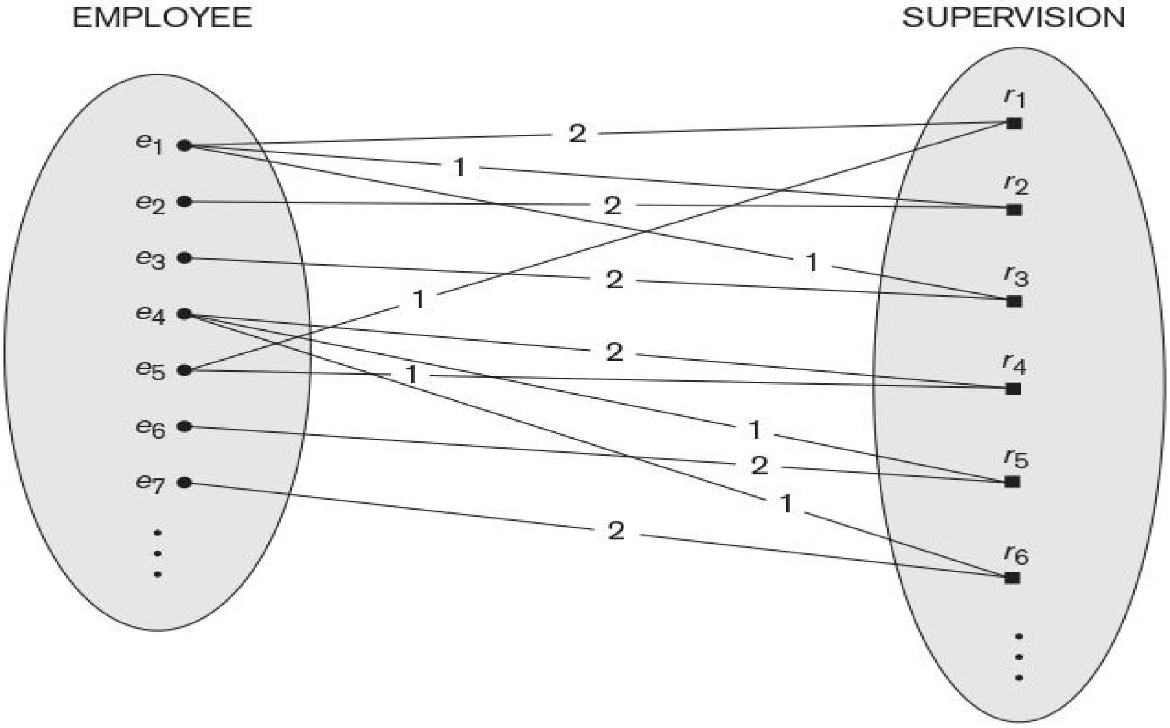
Role Names and Recursive Relationships:

Each entity type that participates in a relationship type plays a particular **role** in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of employee or worker and DEPARTMENT plays the role of department or employer.

Role names are not technically necessary in relationship types where all the participating entity types are distinct. In some cases, the same entity type participates more than once in a relationship type in different roles. Such relationship types are called **recursive relationships**.

Example: The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type participates twice in SUPERVISION: once in the role of supervisor (or boss), and once in the role of supervisee (or subordinate). In Figure 3.11, the lines marked ‘1’ represent the supervisor role, and those marked ‘2’ represent the supervisee role; hence, e1 supervises e2 and e3, e4 supervises e6 and e7, and e5 supervises e1 and e4.

Figure: A recursive relationship SUPERVISION between EMPLOYEE in the supervisor role (1) and EMPLOYEE in the subordinate role (2)



3. Constraints on Binary Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationships represent. In the WORKS_FOR example, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of binary relationship constraints: cardinality ratio and participation.

Cardinality Ratios for Binary Relationships: The cardinality ratio for a binary relationship specifies the maximum number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to any number of employees (N), but an employee can be related to only one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N

Figure: A 1:1 relationship, MANAGES.

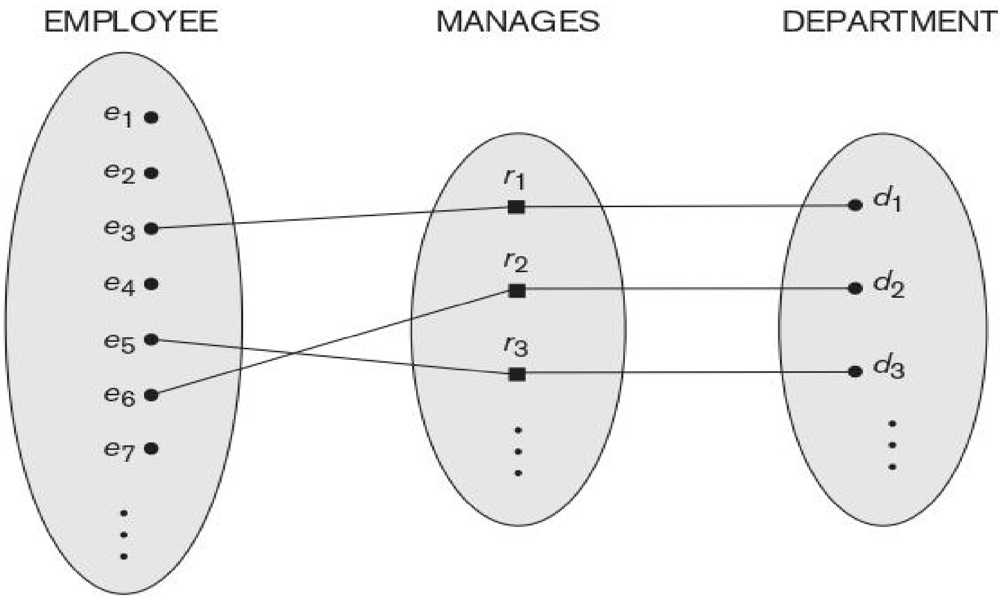
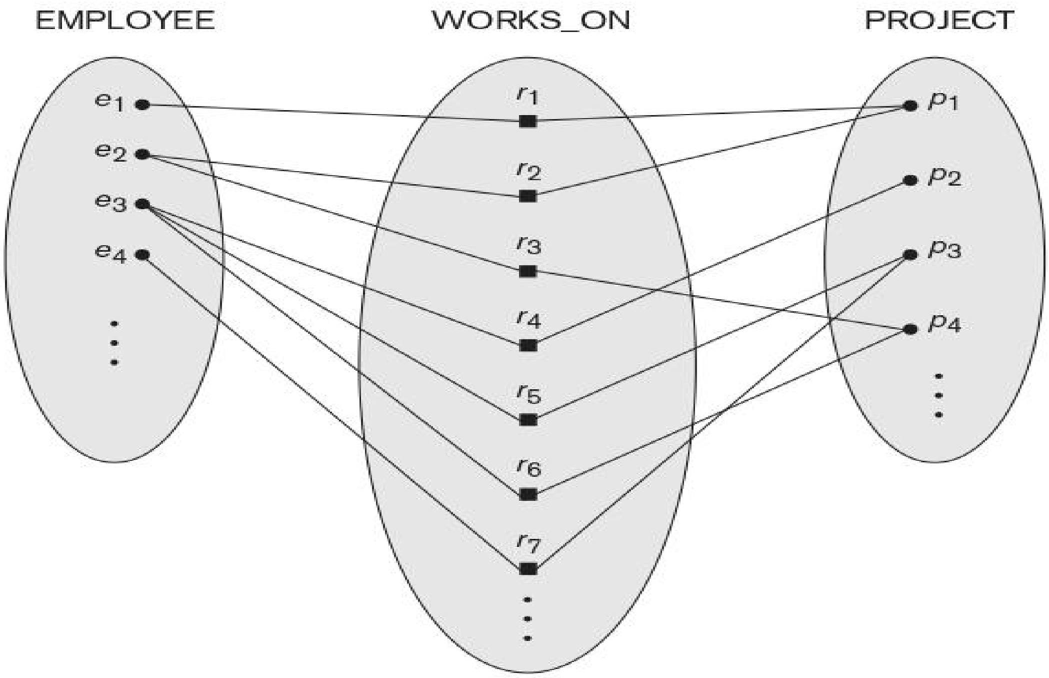


Figure: An M:N relationship, WORKS_ON.



4. Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that a particular employee works on a particular project, we can include an attribute **Hours** for the WORKS_ON relationship type in above Figure.

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types.

For a 1:N relationship type, a relationship attribute can be migrated only to the entity type on the N-side of the relationship.

For M:N (many-to-many) relationship types, some attributes may be determined by the **combination of participating entities** in a relationship instance, not by any single entity. Such attributes must be specified as relationship attributes.

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, Naming Conventions, and Design Issues

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.

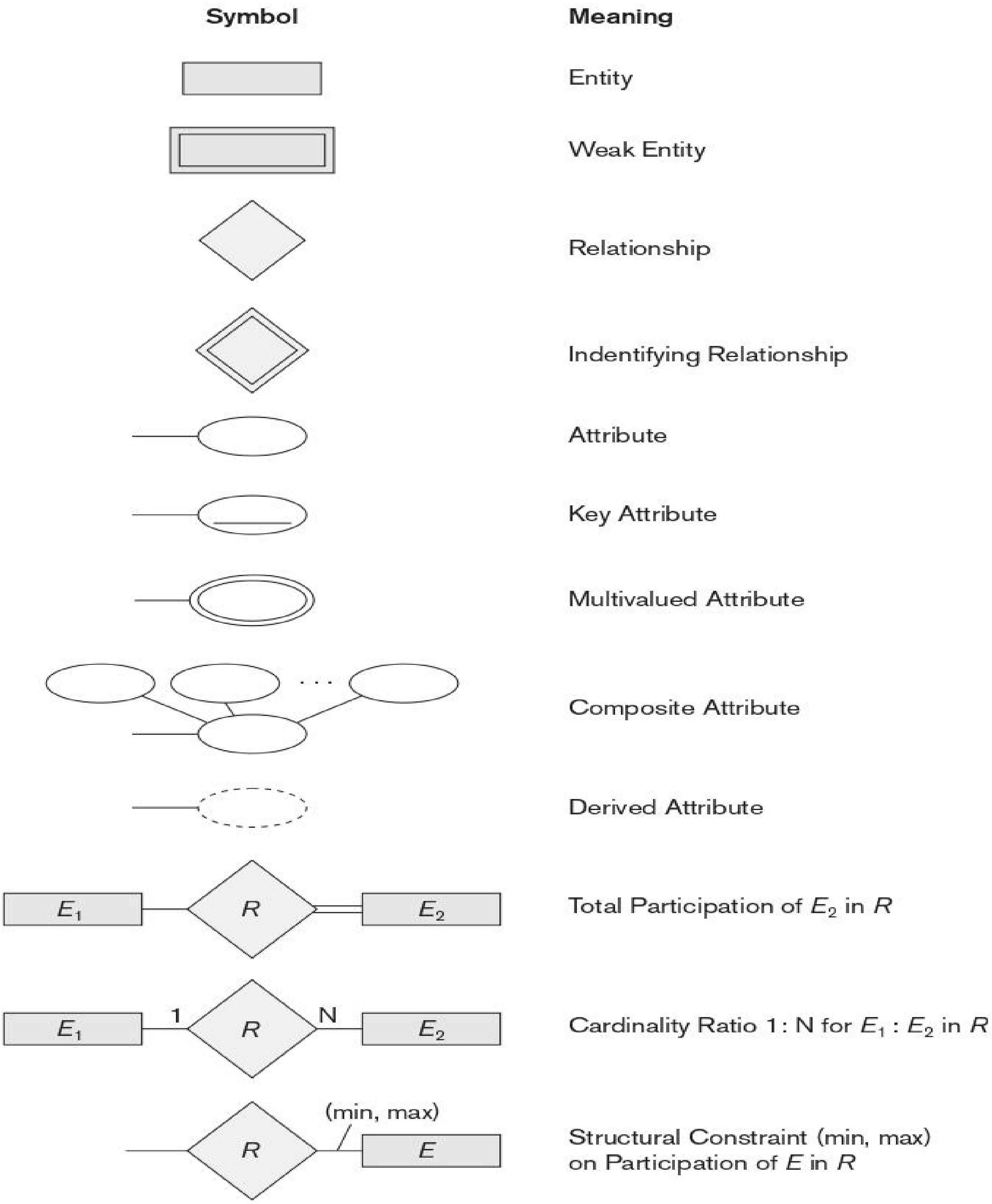
- Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type.

- 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
E_1
R
E_2
Total Participation of E_2 in R
E_1
1
R
N
E_2
Cardinality Ratio 1: N for E_1 : E_2 in R
(min, max)
R
E
Structural Constraint (min, max)
on Participation of E in R

The Enhanced Entity–Relationship (EER) Model

Enhanced Entity–Relationship (EER) Model:

The databases for engineering design and manufacturing (CAD/CAM), telecommunications, complex software systems have more complex than the traditional applications. This led to the development of additional semantic data modeling concepts that were incorporated into conceptual data models such as the ER model.

The additional semantic data modeling concepts are

1. Subclasses, Superclasses, and Inheritance
2. Specialization and Generalization
3. Various types of Constraints on Specialization and Generalization
4. Modeling of UNION Types Using Categories

1. Subclasses, Superclasses, and Inheritance: An entity type has numerous sub groupings of its entities that are meaningful and need to be represented explicitly.

For example, the entity type EMPLOYEE may be grouped further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on.

Each of these sub groupings, a subclass of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** for each of these subclasses. The following Figure shows how to represent these concepts diagrammatically in EER diagrams.

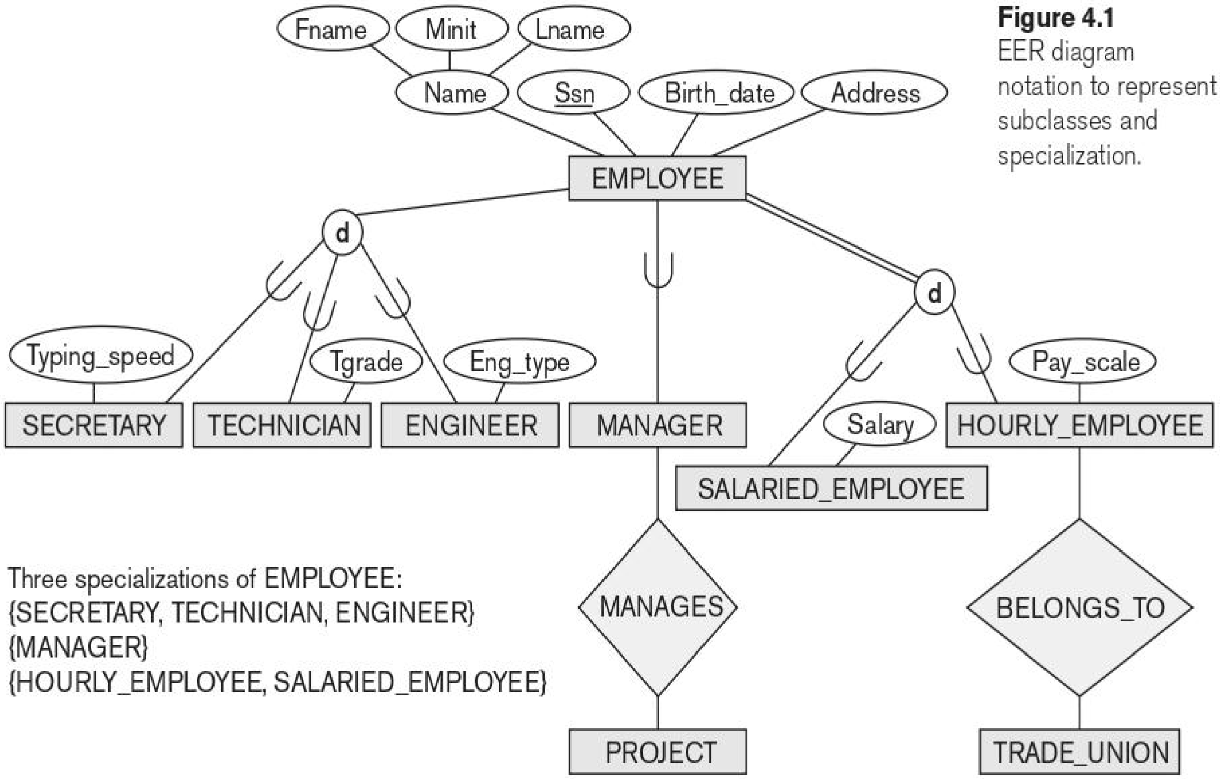


Figure 4.1
EER diagram notation to represent subclasses and specialization.

Three specializations of EMPLOYEE:
{SECRETARY, TECHNICIAN, ENGINEER}
{MANAGER}
{HOURLY_EMPLOYEE, SALARIED_EMPLOYEE}

The relationship between a superclass and any one of its subclasses, a superclass/subclass or supertype/subtype or simply class/subclass relationship.

For example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships.

An important concept associated with subclasses is that of type **inheritance**. An entity that is a member of a subclass inherits all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates

2. Specialization and Generalization

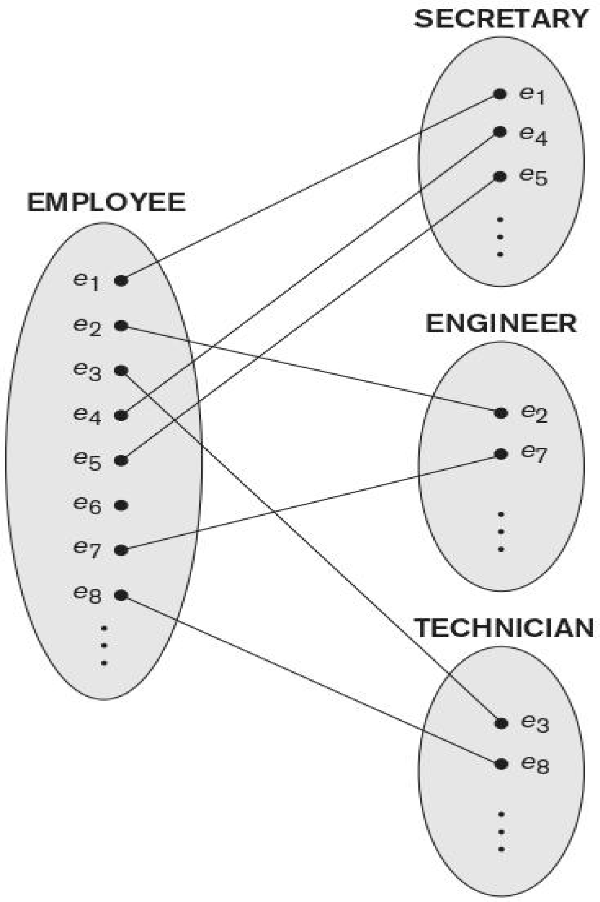
Specialization:

Specialization is the process of defining a set of subclasses of an entity type. This entity type is called the superclass of the specialization.

For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE. The super class EMPLOYEE identifies subclass entities based on the job type of the employee.

The following Figure shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization.

Figure: Instances of a specialization.



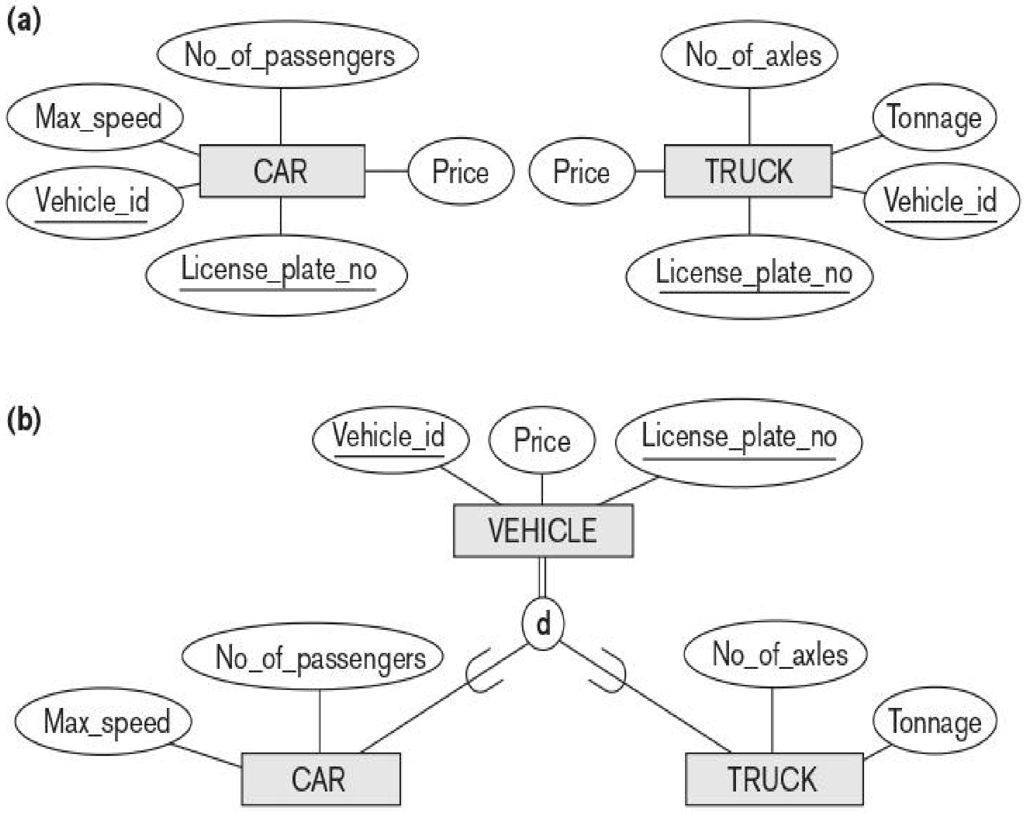
A superclass/subclass relationship such as EMPLOYEE/SECRETARY is a 1:1 relationship two distinct entities are related. In a superclass/subclass relationship the entity in the subclass is the same as entity in the superclass but is playing a specialized role.

For example, an EMPLOYEE specialized in the role of SECRETARY or an EMPLOYEE specialized in the role of TECHNICIAN.

Generalization: Generalization is a reverse process of abstraction in which we suppress the differences among several entity types, identify their common features, and generalize them into a single superclass of which the original entity types are special subclasses. Generalization refers to the process of defining a generalized entity type from the given entity types.

For example, consider the entity types CAR and TRUCK shown in Figure. Because they have several common attributes, they can be generalized into the entity type VEHICLE. Both CAR and TRUCK are now subclasses of the generalized superclass VEHICLE.

Generalization. (a) Two entity types, CAR and TRUCK.
(b) Generalizing CAR and TRUCK into the superclass VEHICLE.

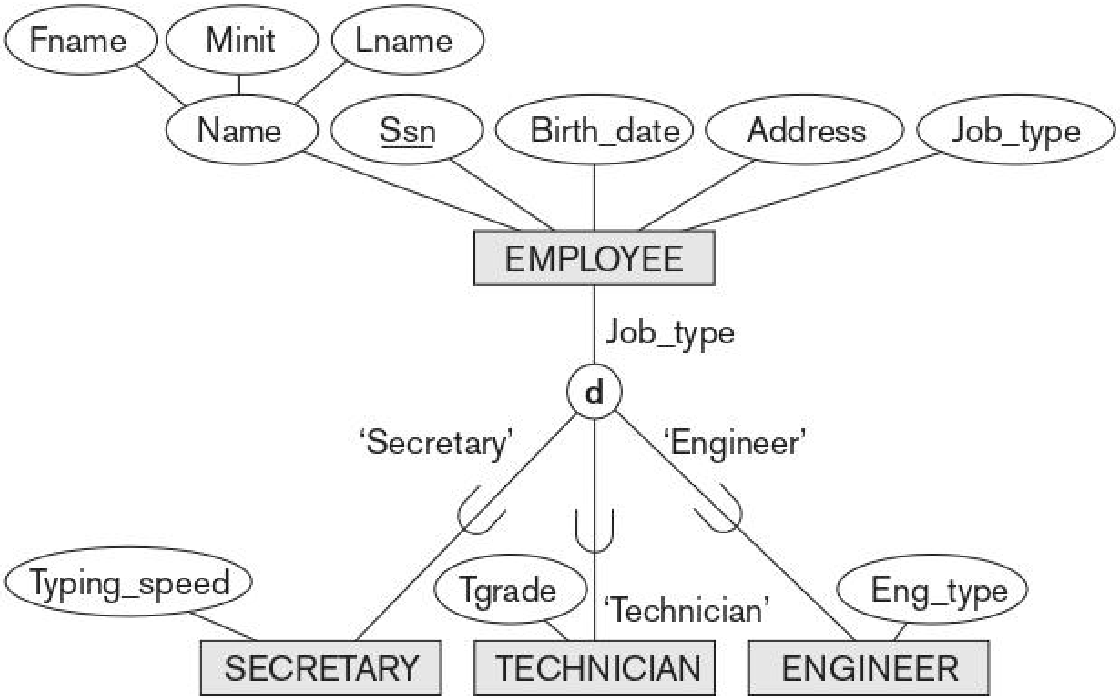


3. Constraints and Characteristics of Specialization and Generalization Hierarchies

Constraints on Specialization and Generalization: A specialization consists of number of sub classes. In such cases we use circle notation. A specialization consists of single subclass. In such cases we do not use the circle notation. In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called predicate-defined subclasses.

For example, if the EMPLOYEE entity type has an attribute Job_type, as shown in Figure, we can specify the condition in the SECRETARY subclass by the condition (Job_type = 'Secretary'). This condition is a constraint specifying that exactly.

Figure: EER diagram notation for an attribute-defined specialization on Job_type



If all subclasses in a specialization have their membership condition on the same attribute of the superclass, the specialization itself is called an **attribute-defined specialization**, and the attribute is called the **defining attribute of the specialization.**

Two other constraints may apply to a specialization. They are he **disjointness constraint, and completeness constraint**.

Disjointness constraint: An entity can be a member of at most one of the subclasses of the specialization. This is displayed by placing a 'd' in the circle. An entity may be a member of more than one sub class of specialiazation. This is displayed by placing 'o' in the circle.

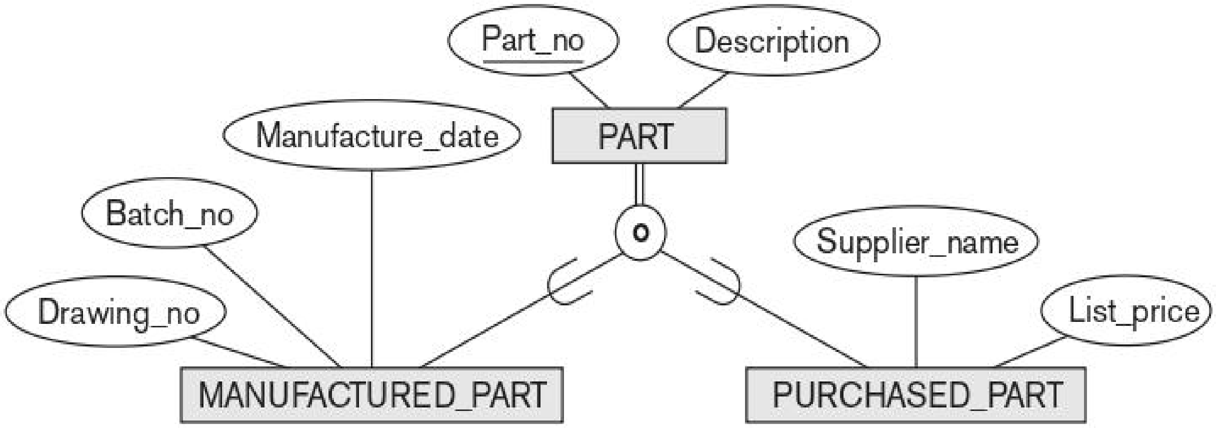
Completeness (or totalness) constraint: This completeness constraint may be total or partial. A total specialization constraint specifies that every entity in the superclass must be a member of at least one subclass in the specialization.

For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} is a total specialization of EMPLOYEE.

The total specialization is represented in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a partial specialization, which allows an entity not to belong to any of the subclasses.

For example, if some EMPLOYEE entities do not belong to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} in Figures then that specialization is partial.

Figure: EER diagram notation for an overlapping (nondisjoint) specialization.



The disjointness and completeness constraints are independent. Hence, we have the following four possible constraints on a specialization:

- ■ Disjoint, total
- ■ Disjoint, partial
- ■ Overlapping, total
- ■ Overlapping, partial.

Specialization and Generalization Hierarchies and Lattices:

A subclass itself may have further subclasses, forming a hierarchy or a lattice of specializations. A specialization hierarchy has the constraint that every subclass participates as a subclass in only one class/subclass relationship; that is, each subclass has only one parent. A specialization lattice, a subclass can be a subclass in more than one class/subclass relationship.

The following diagram shows hierarchical lattices.

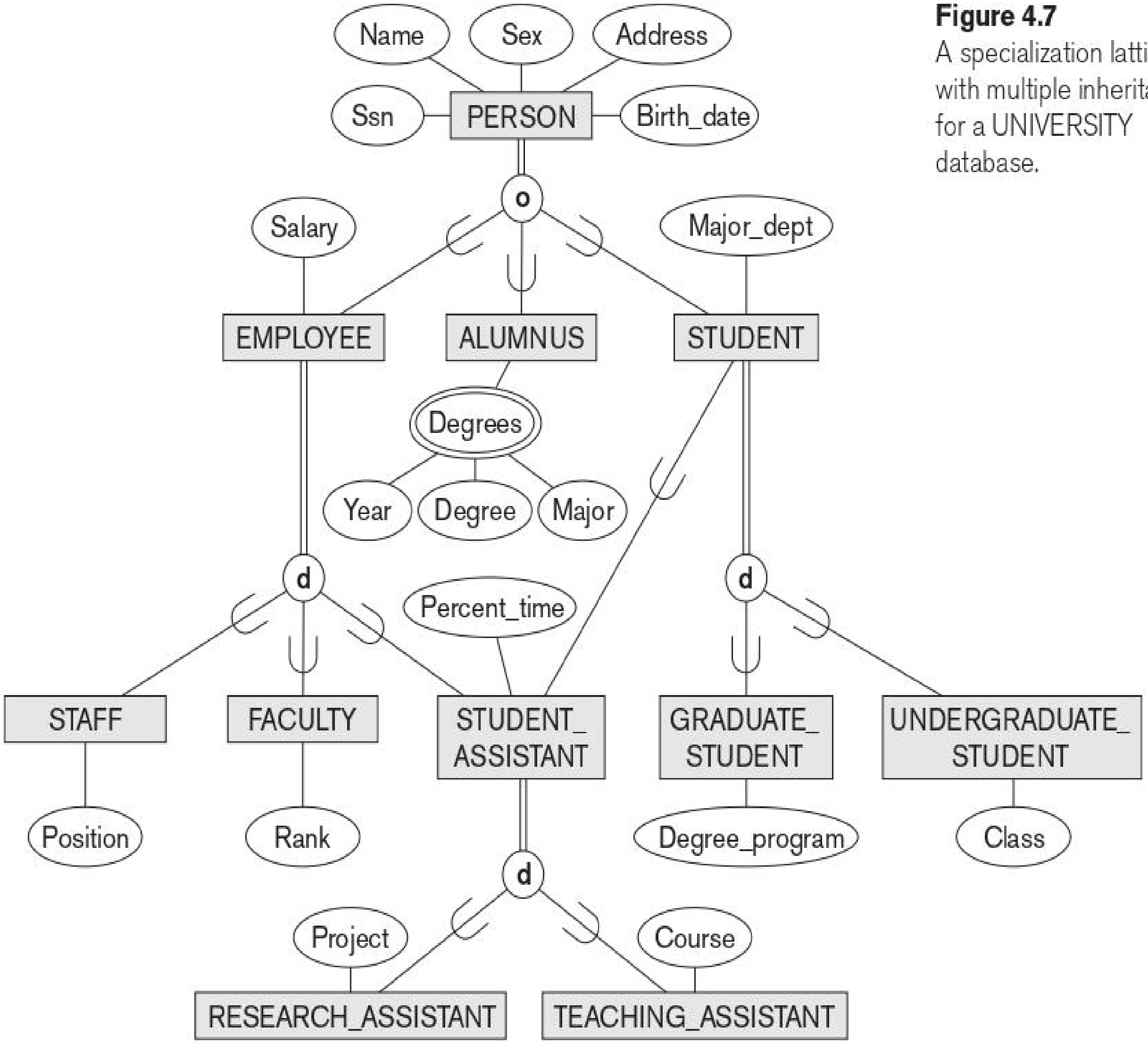


Figure 4.7
A specialization lattice with multiple inheritance for a UNIVERSITY database.

The PERSON entity type is specialized into the subclasses {EMPLOYEE, ALUMNUS, STUDENT}. This specialization is overlapping.

The sub class EMPLOYEE is the superclass for the specialization {STUDENT_ASSISTANT, FACULTY, STAFF}.

The subclass STUDENT is the superclass for the specialization {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT},

Finally, the sub class STUDENT_ASSISTANT is the superclass for the specialization into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}.

The subclass with more than one superclass is called a **shared subclass**, leads to lattices, if no shared sub classes leads to hierarchy.

In the university database the shared sub class in class STUDENT_ASSISTANT which inherits attributes from both EMPLOYEE and STUDENT inherit the same attributes from PERSON. If an attribute in the super class is inherited more than once via different paths in the lattice, then the attributes are included only once in the shared sub classes.

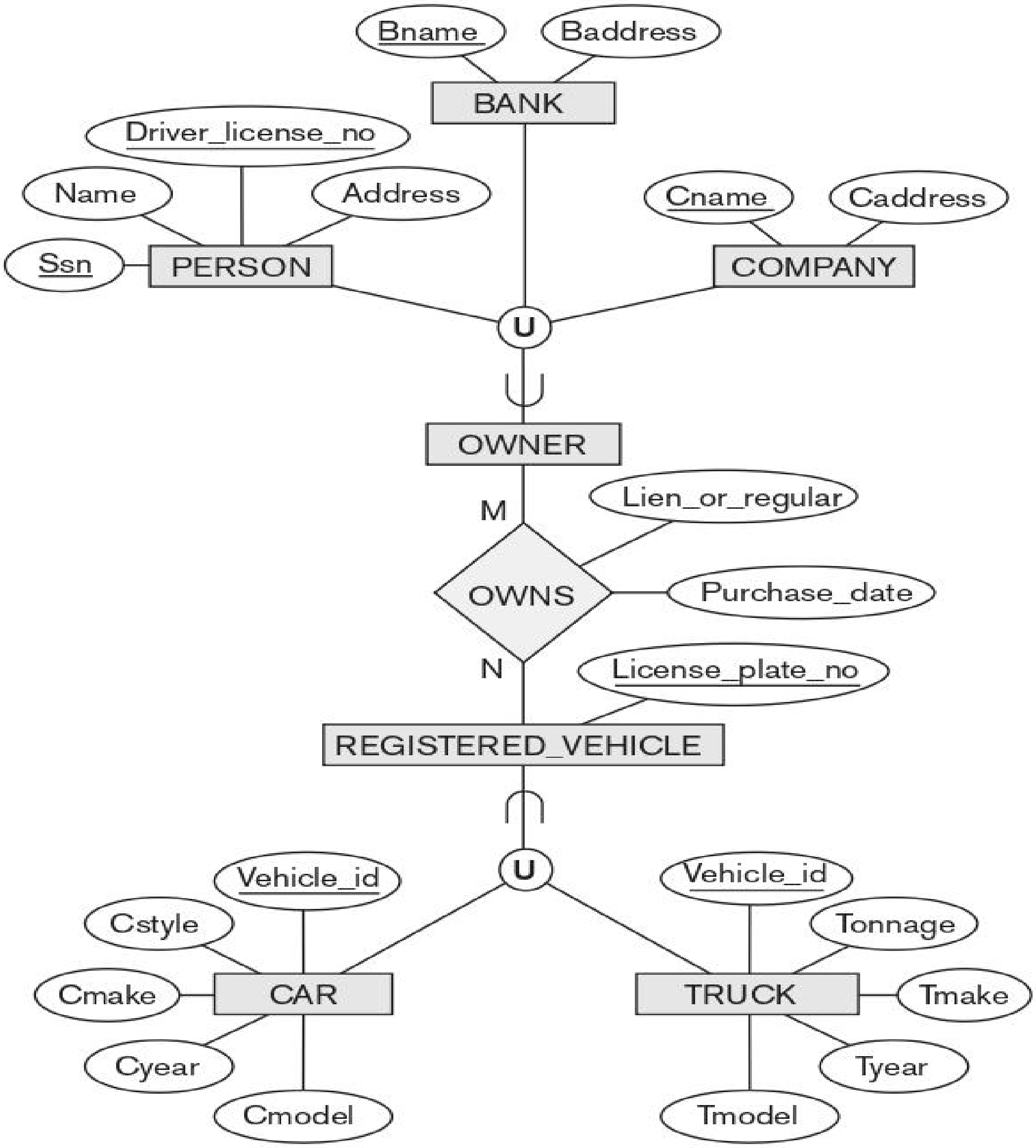
4. Modeling of UNION Types Using Categories

It is sometimes necessary to represent a collection of entities from different entity types. In this case, a subclass will represent a collection of entities that is a subset of the UNION of entities from distinct entity types; we call such a subclass a union type or a category.

For example, consider three entity types: PERSON, BANK, and COMPANY. In a database for vehicle registration, an owner of a vehicle can be a person, a bank or a company. We need to create a class that includes entities of all three types to play the role of vehicle owner.

A category OWNER that is a subclass of the UNION of the three entity sets of COMPANY, BANK, and PERSON is created for this purpose. We display categories in an EER diagram as shown below.

Figure: Two categories (union types): OWNER and REGISTERED_VEHICLE.



The superclasses COMPANY, BANK, and PERSON are connected to the circle with the 'U' symbol, which stands for the set union operation.

A category such as REGISTERED_VEHICLE in Figure implies that only cars and trucks, but not other types of entities, can be members of REGISTERED_VEHICLE. A category can be total or partial. A total category holds the union of all entities in its superclasses, whereas a partial category can hold a subset of the union. A total category is represented by a double line connecting the category and the circle, whereas a partial category is indicated by a single line.

The UNIVERSITY Database Example

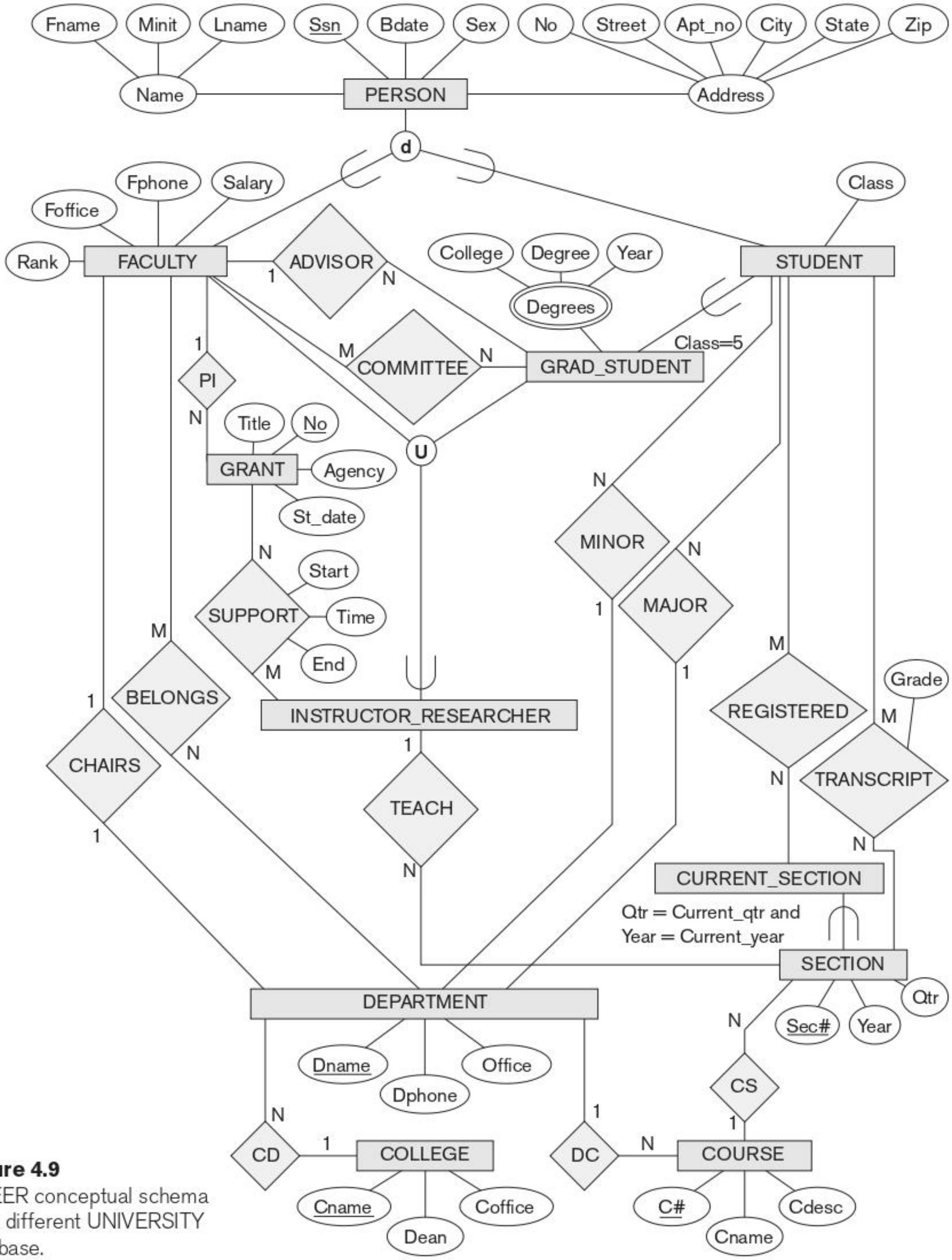


Figure 4.9
An EER conceptual schema for a different UNIVERSITY database.

The Relational Data Model and Relational Database Constraints

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.

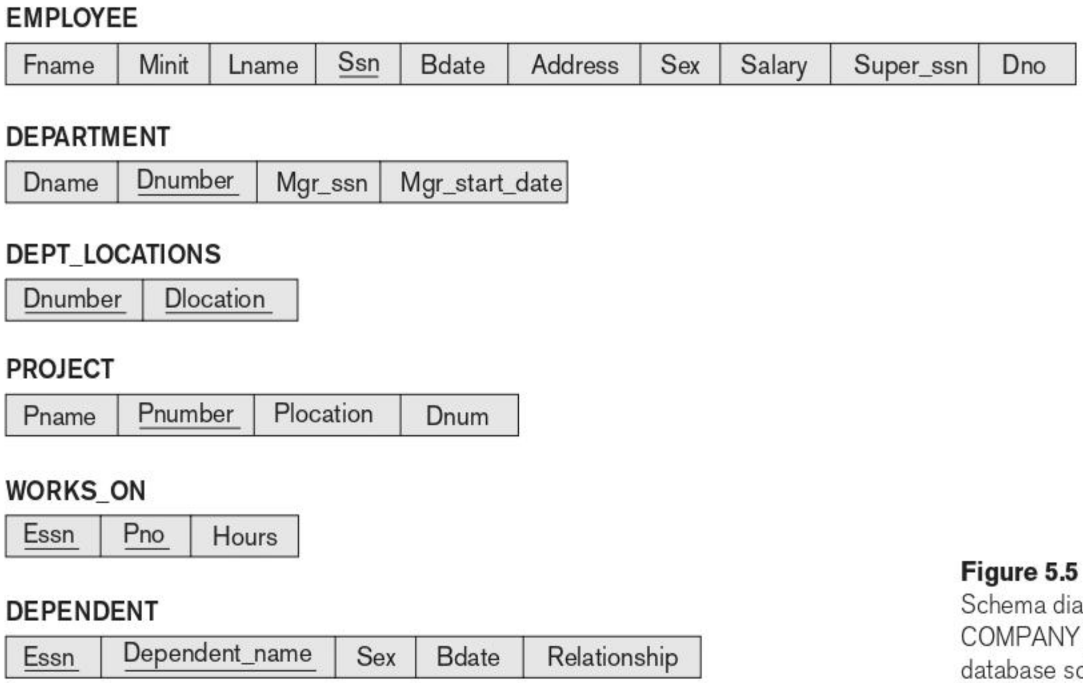
For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then 'Name' of STUDENT is constrained to be NOT NULL.

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.

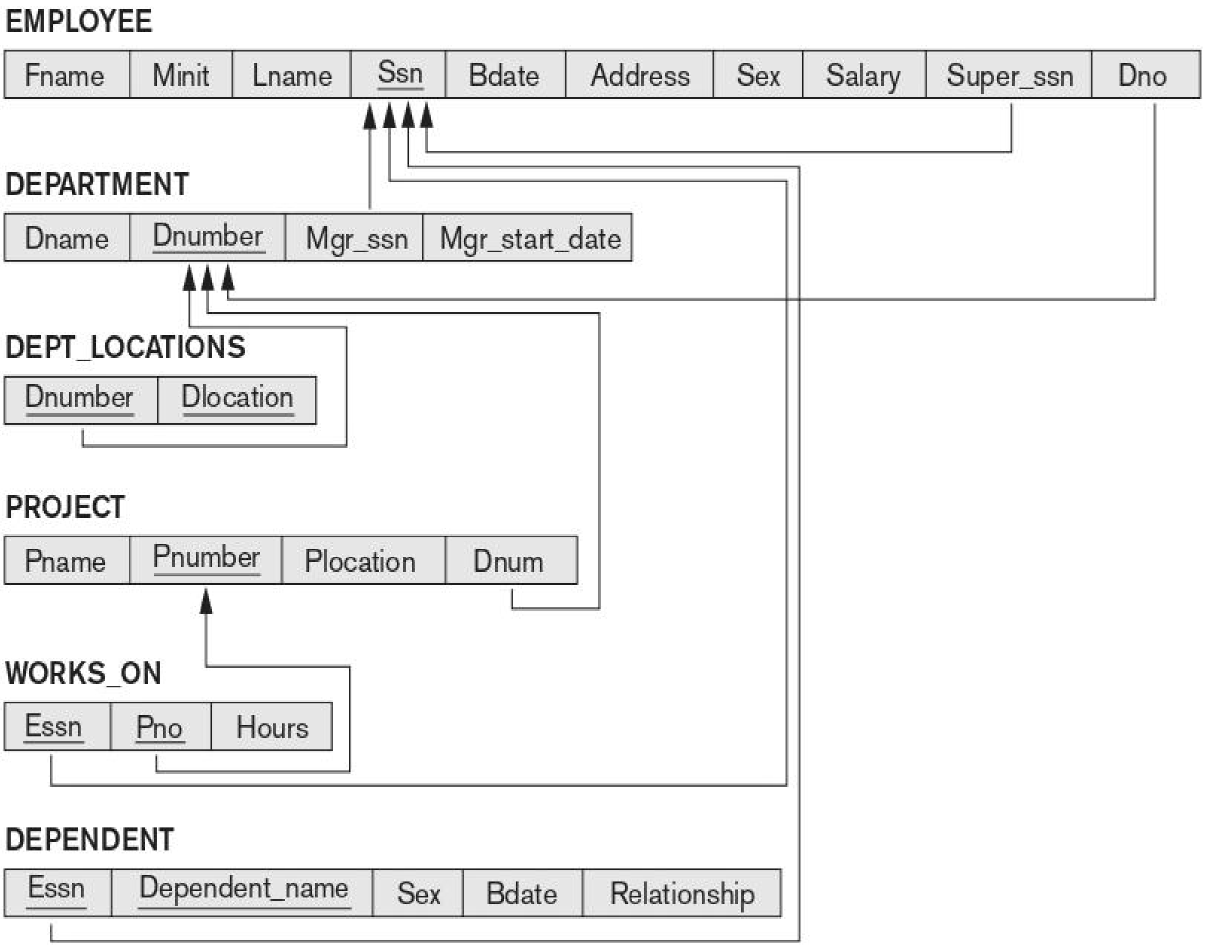
To define referential integrity more formally, we define the concept of a **foreign key.** The concept of foreign key is to specify a referential integrity constraints between the two relation schemas R1 and R2.

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

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.

****

UNIT - II

Relational Algebra and Relational Calculus

Relational Algebra

The basic set of operations for the formal relational model is the relational algebra. These operations enables user to specify basic retrieval request. The result of a retrieval query is a new relation, which formed one or more relations.

The relational operations can be divided into two groups. One group includes **set operations.** The Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group include SELECT, PROJECT, and JOIN.

SELECT and PROJECT operations are unary operations that operate on one relation. JOIN and other operations are binary operations which operates on two tables.

Unary Relational Operations

Unary Relational Operations: SELECT and PROJECT

The SELECT Operation:

The SELECT operation selects tuples that satisfy a given condition. We use the lowercase Greek letter (σ) to denote selection. The condition (predicate) appears as a subscript to σ. The argument 'relation' is given parenthesis following to σ.

In general the SELECT operation is denoted by

$\boldsymbol{\sigma}_{\textbf{<selection condition>}} \textbf{(R)}$

The comparison (<, >, ≤, ≥, =, ≠) and logical (AND, OR, NOT) operators are allowed in conditions.

Example:

1. Select the EMPLOYEE tuples whose dept is 4

   $\boldsymbol{\sigma}_{\textbf{Dno=4}}\textbf{(EMPLOYEE)}$

2. Select the EMPLOYEE tuples whose salary is greater than 30,000

   $\sigma_{\text{Salary}>30000}(\text{EMPLOYEE})$

3. Select the tiples for all employees who either work in dept 4 and salary > 25000 per year or work in dept 5 and salary > 30000

$$\sigma_{(\text{Dno}=4 \text{ AND Salary}>25000) \text{ OR } (\text{Dno}=5 \text{ AND Salary}>30000)}(\text{EMPLOYEE})$$

In SQL, the SELECT condition is typically specified in the WHERE clause of a query. For example, the following operation:

$$\sigma_{\text{Dno}=4 \text{ AND Salary}>25000}\ (\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT    *
FROM   EMPLOYEE
WHERE   Dno=4  AND  Salary>25000;
```

The PROJECT Operation:

The PROJECT operation selects certain columns from the relation and other columns are discarded, we used the Greek letter ‘π’ to denote the projection. The column names appears as a subscript of ‘π’, these column names appear in the result. The argument ‘relation’ is given in the parenthesis following the ‘π’.

Syntax: The general form of the PROJECT operation is

$$\pi_{(\text{attribute list})}(\text{Relation})$$

For example, to list each employee’s first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$$

For example, to retrieve the firstname, lastname and salary of all employee who works in department number 5.

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

We can explicitly show the sequence of operations, giving a name to each intermediate relation, and using the assignment operation, denoted by ← (left arrow), as follows:

$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$

To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$\text{TEMP} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$

$\text{R(First\_name, Last\_name, Salary)} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{TEMP})$

The RENAME operation has 3 forms as shown below:

$\rho_{S(B1,\, B2,\, \dots,\, Bn)}(R)$ or $\rho_S(R)$ or $\rho_{(B1,\, B2,\, \dots,\, Bn)}(R)$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name and B1, B2,….Bn are new attribute names.

Relational Algebra Operations from Set Theory

The UNION, INTERSECTION, and MINUS Operations:

These are the binary operations. These operations will take two relations as input and produce one relation as output.

UNION:

The UNION operation is denoted by 'U'. The result of 'R ∪ S' is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

Ex: To retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$

$\text{RESULT1} \leftarrow \pi_{\text{Ssn}}(\text{DEP5\_EMPS})$

$\text{RESULT2(Ssn)} \leftarrow \pi_{\text{Super\_ssn}}(\text{DEP5\_EMPS})$

$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$

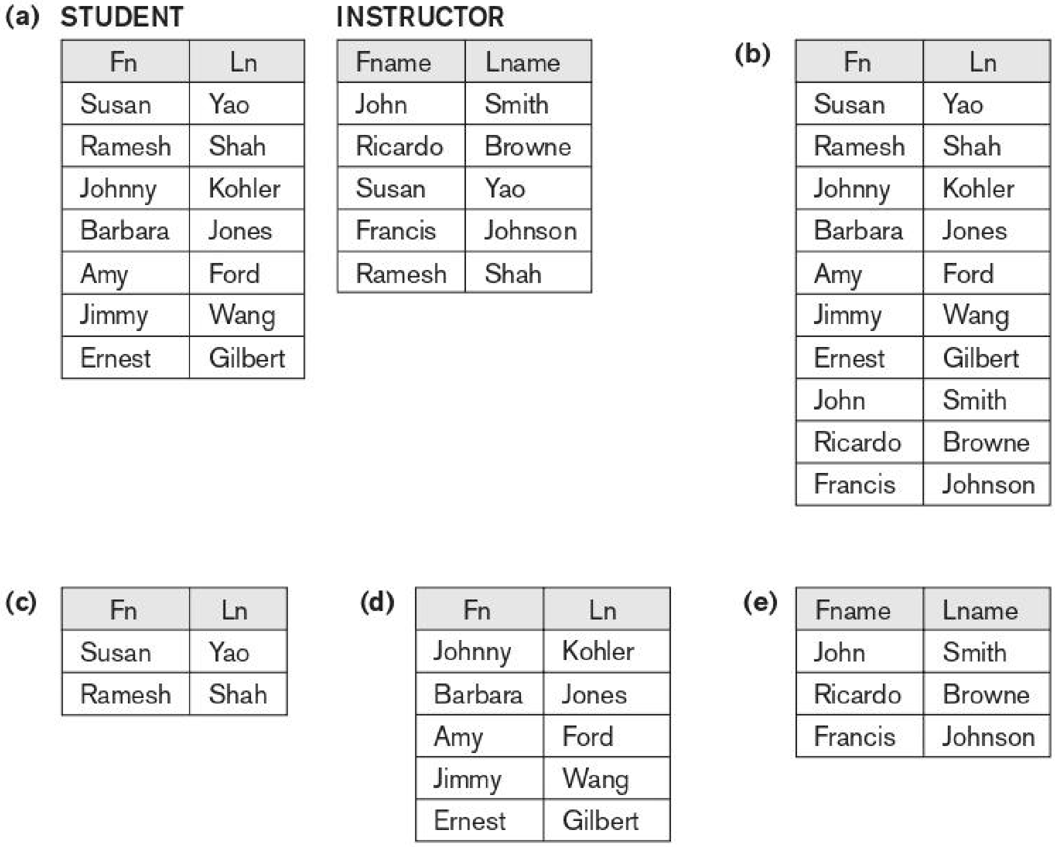
INTERSECTION:

The INTERSECTION operation is denoted by '∩'. The result of R ∩ S, is a relation that includes all tuples that are in both R and S.

SET DIFFERENCE (or MINUS):

The DIFFERENCE operation is denoted by '-'. The result of R – S, is a relation that includes all tuples that are in 'R' but not in 'S'.

Figure 8.4
The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.



(a) STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

Notice that both UNION and INTERSECTION are commutative operations; that is,

$$\mathbf{R \cup S = S \cup R \text{ and } R \cap S = S \cap R}$$

Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are also associative operations; that is,

$$\mathbf{R \cup (S \cup T) = (R \cup S) \cup T \text{ and } (R \cap S) \cap T = R \cap (S \cap T)}$$

The MINUS operation is not commutative; that is, in general,

$$\mathbf{R - S \neq S - R}$$

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

The CARTESIAN PRODUCT operation is denoted by '**×**'. The result of R × S is a relation that include new element by combining every tuple from relation R with every tuple from relation S.

Ex: To retrieve a list of names of each female employee dependents.

$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex='F'}}(\text{EMPLOYEE})$

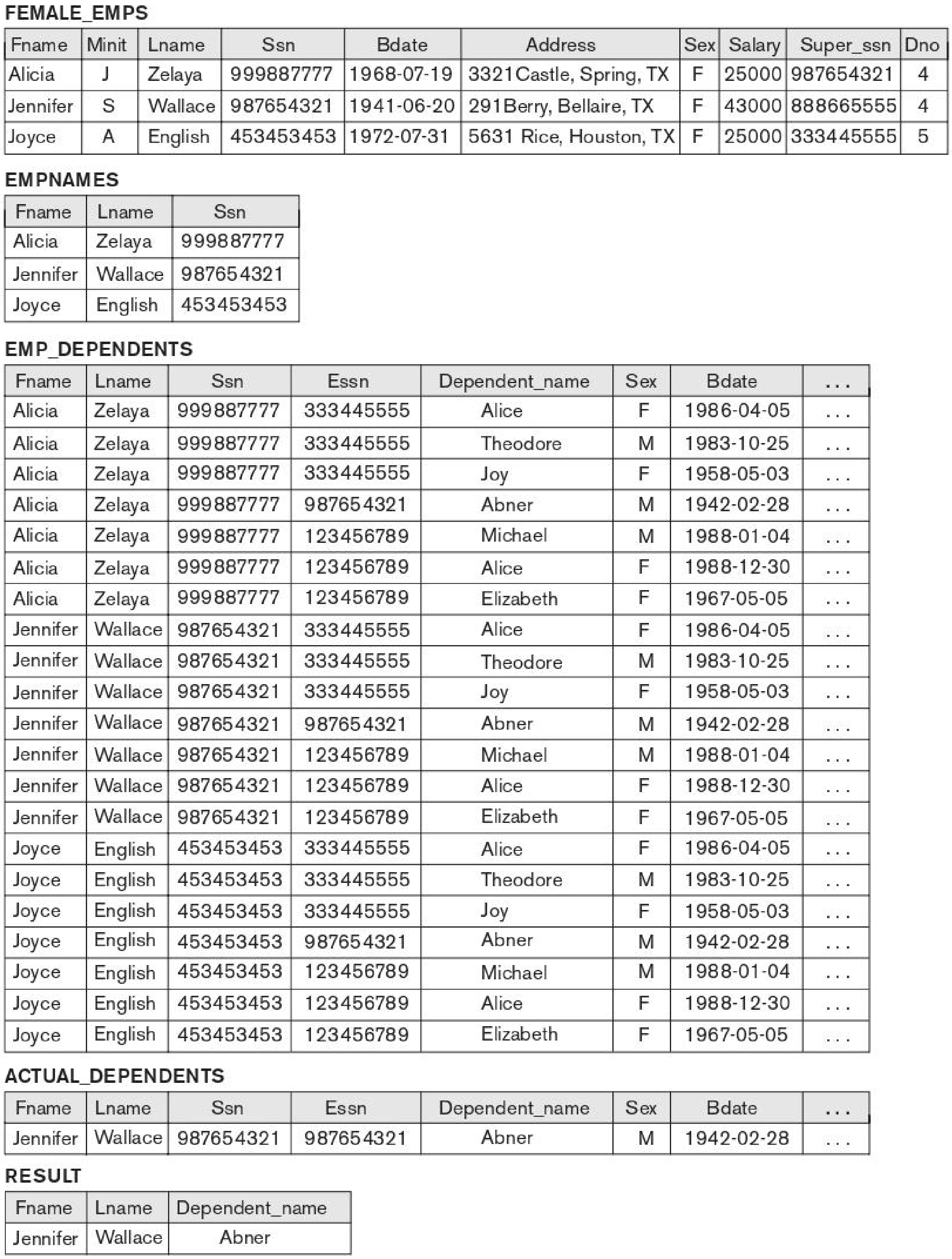
$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$

$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$

$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$

$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$

Figure 8.5
The CARTESIAN PRODUCT (CROSS PRODUCT) operation.



FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	. . .
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	. . .
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	. . .
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	. . .
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	. . .
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	. . .
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	. . .
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	. . .
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	. . .
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	. . .
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	. . .
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	. . .
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	. . .
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	. . .
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	. . .
Joyce	English	453453453	333445555	Alice	F	1986-04-05	. . .
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	. . .
Joyce	English	453453453	333445555	Joy	F	1958-05-03	. . .
Joyce	English	453453453	987654321	Abner	M	1942-02-28	. . .
Joyce	English	453453453	123456789	Michael	M	1988-01-04	. . .
Joyce	English	453453453	123456789	Alice	F	1988-12-30	. . .
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	. . .

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	. . .
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	. . .

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Binary Relational Operations

Binary Relational Operations: JOIN and DIVISION

JOIN Operation:

The JOIN operation is denoted by ‘ ’, is used to combine related tuples from two relations into single tuples.

The general form of a JOIN operation on two relations[5] $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is

$$R \bowtie_{\text{<join condition>}} S$$

Ex: Get the manager’s name

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$$
$$\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$$

Variationsof JOIN: The EQUIJOIN and NATURAL JOIN

EQUIJOIN: The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an EQUIJOIN.

$$R \bowtie_{\text{<join condition>}} S$$

NATURAL JOIN:

The standard definition of NATURAL JOIN requires that the two join attributes have the same name in both relations. If this is not the case, a renaming operation is applied first.

Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

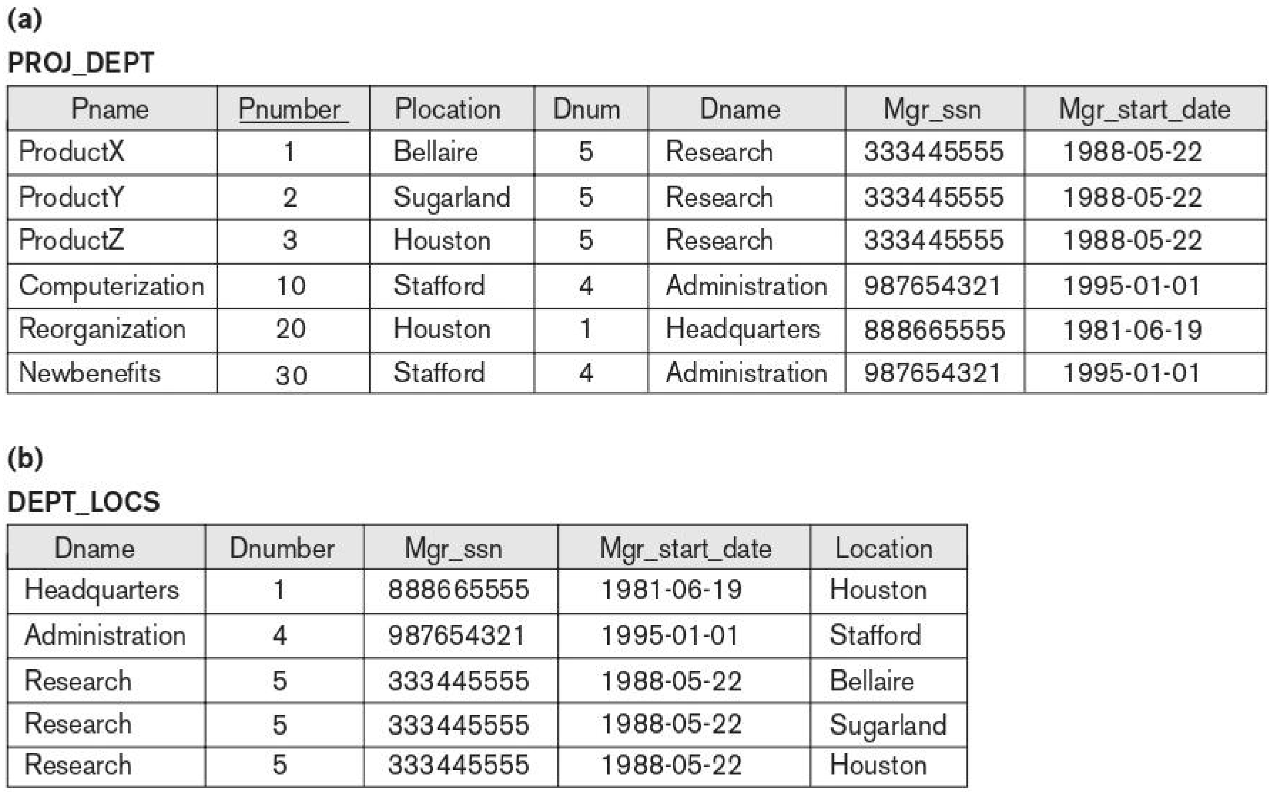
PROJ_DEPT ← PROJECT * $\rho_{(\text{Dname, Dnum, Mgr\_ssn, Mgr\_start\_date})}$(DEPARTMENT)

The same query can be done in two steps by creating an intermediate table DEPT as follows:

DEPT ← $\rho_{(\text{Dname, Dnum, Mgr\_ssn, Mgr\_start\_date})}$(DEPARTMENT)
PROJ_DEPT ← PROJECT * DEPT

If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS



(a)

PROJ_DEPT

Pname	Pnumber	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

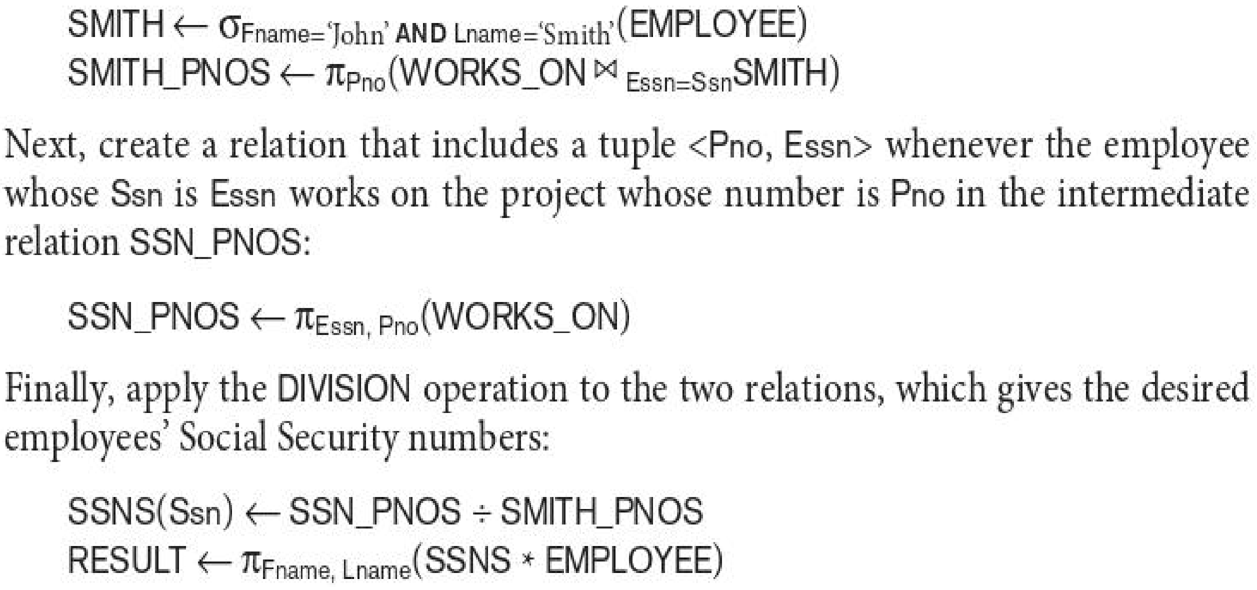
Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 8.7
Results of two natural join operations. (a) proj_dept ← project * dept.
(b) dept_locs ← department * dept_locations.

The DIVISION Operation:

The DIVISION operation, denoted by ÷. The DIVISION operation is applied to two relations $R(Z) \div S(X)$, where the attributes of S are a subset of the attributes of R; that is, $X \subseteq Z$. The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R [Y] = t$, and with $t_R [X] = t_S$ for every tuple t_S in S.

Ex: To retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:



$$\text{SMITH} \leftarrow \sigma_{\text{Fname='John' AND Lname='Smith'}}(\text{EMPLOYEE})$$
$$\text{SMITH\_PNOS} \leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} \bowtie_{\text{Essn=Ssn}} \text{SMITH})$$

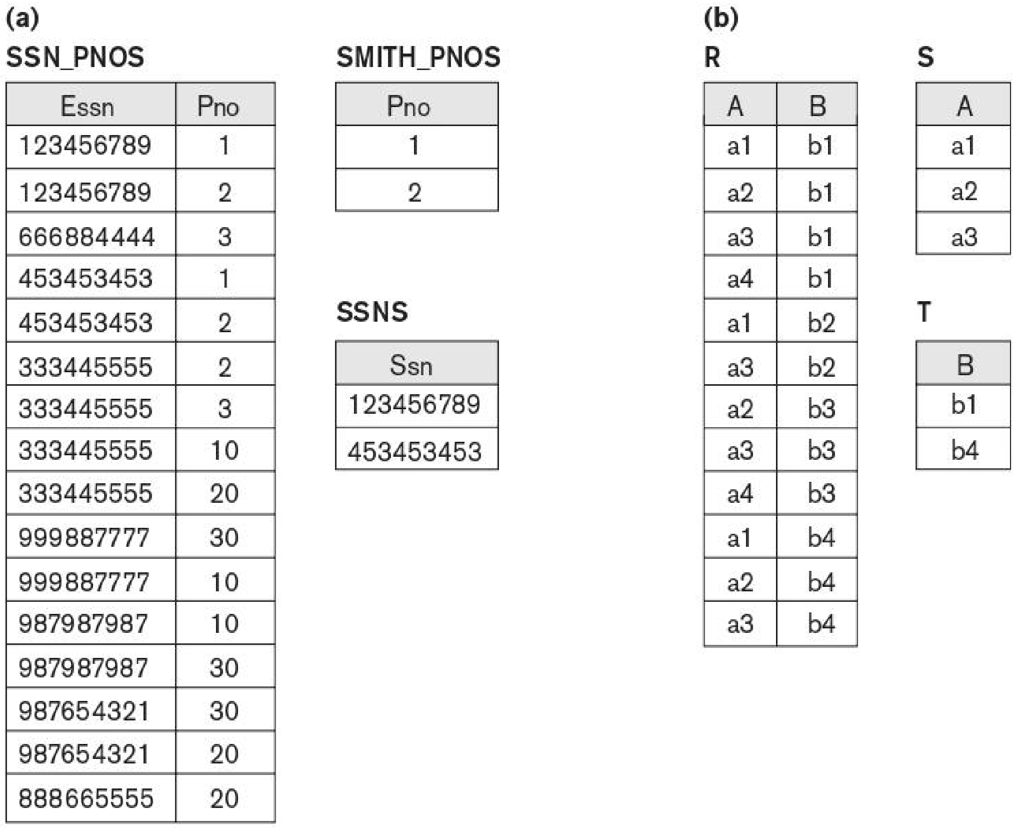
Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$\text{SSN\_PNOS} \leftarrow \pi_{\text{Essn, Pno}}(\text{WORKS\_ON})$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$\text{SSNS(Ssn)} \leftarrow \text{SSN\_PNOS} \div \text{SMITH\_PNOS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname}}(\text{SSNS} * \text{EMPLOYEE})$$

Figure 8.8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.



(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

Additional Relational Operations

Generalized Projection: The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list.

The generalized form can be expressed as:

$$\pi_{F1,\ F2,\ \ldots,\ Fn}\,(R)$$

where F1, F2, … , Fn are functions over the attributes in relation R and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result.

As an example, consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary – Deduction,

Bonus = 2000 * Years_service, and

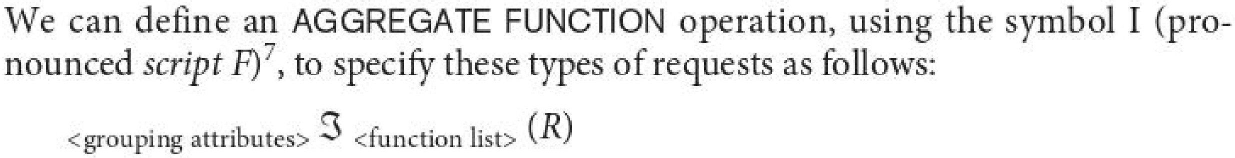
Tax = 0.25 * Salary

Then a generalized projection combined with renaming may be used as follows:

REPORT ← ρ(Ssn, Net_salary, Bonus, Tax)(πSsn, Salary – Deduction, 2000 * Years_service, 0.25 * Salary(EMPLOYEE))

Aggregate Functions and Grouping:

The basic relational algebra is to specify mathematical aggregate functions on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.



We can define an AGGREGATE FUNCTION operation, using the symbol I (pronounced *script F*)[7], to specify these types of requests as follows:

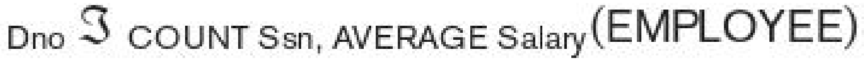
$$_{\text{<grouping attributes>}}\Im_{\text{<function list>}}(R)$$

where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of <function> <attribute> pairs. The following functions such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT are allowed in each pair and <attribute> in relation R.

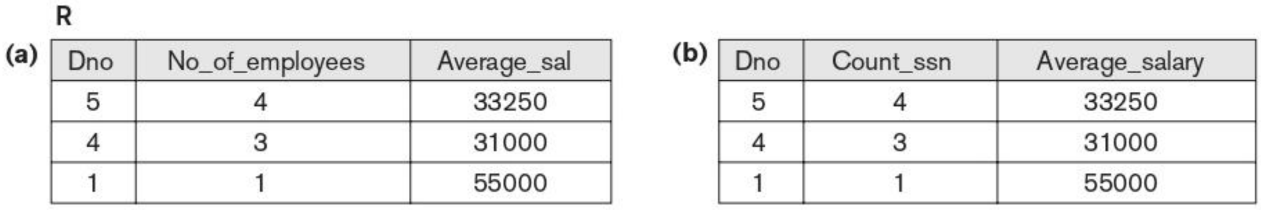
Ex: 1. To retrieve each department number, the number of employees in the department, and their average salary.

$$\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}\ (_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}\ (\text{EMPLOYEE}))$$

2. Find the total number of employees and their average salary.



$$_{\text{Dno}}\,\Im\,_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$$



R

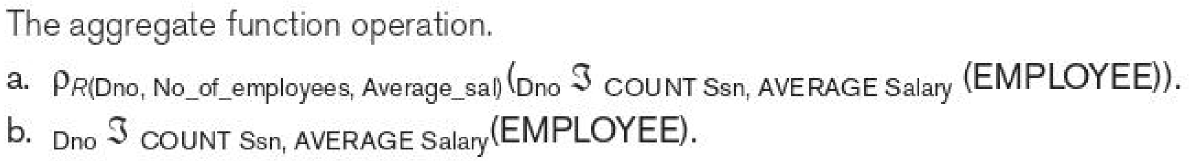
(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

Figure:



The aggregate function operation.

a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}(_{\text{Dno}}\,\Im\,_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE}))$.

b. $_{\text{Dno}}\,\Im\,_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

OUTER JOIN Operations:

The JOIN operation that are necessary to specify certain types of queries. In NATURAL JOIN operation R * S, only tuples from R that have matching tuples in S and vice versa appear in the result.

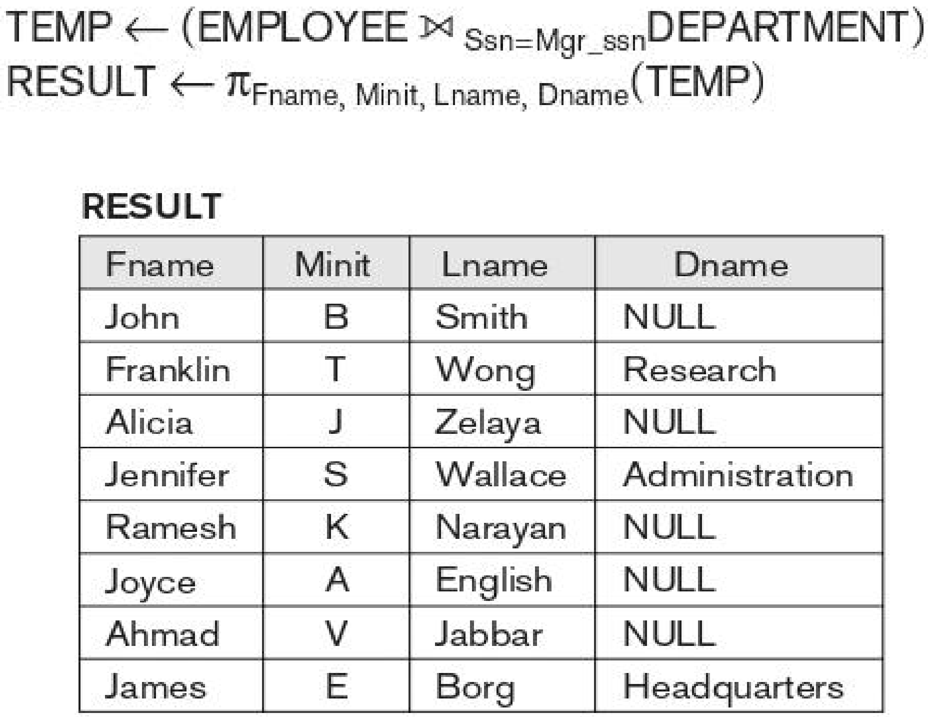
Tuples with NULL values in the join attributes are also eliminated.

A set of operations, called outer joins, were developed for the case where the user wants to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN.

LEFT OUTER JOIN:

A LEFT OUTER JOIN is denoted by , keep every tuple in first or left relation R in the result of R S . If no matching tuples is found in S, then the attributes of S in the join result are filled with NULL values.

Ex: List of employee names and also name of departments, if they do not manage one we can indicate it with NULL value,



$$TEMP \leftarrow (EMPLOYEE \rtimes_{Ssn=Mgr\_ssn} DEPARTMENT)$$
$$RESULT \leftarrow \pi_{Fname,\ Minit,\ Lname,\ Dname}(TEMP)$$

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

RIGHT OUTER JOIN:

The RIGHT OUTER JOIN, denoted by , keeps every tuple in the second, or right, relation S in the result of R S. If no matching tuple is found in R, then the attributes of R in the join result are filled with NULL values.

FULL OUTER JOIN: The full outer join denoted by , keeps all tuples in both the left and the right relations when no matching tuples are found, padding with NULL values.

Recursive operation: This operation is applied in between tuples of same type. For ex, the relationship between an employee and supervisor.

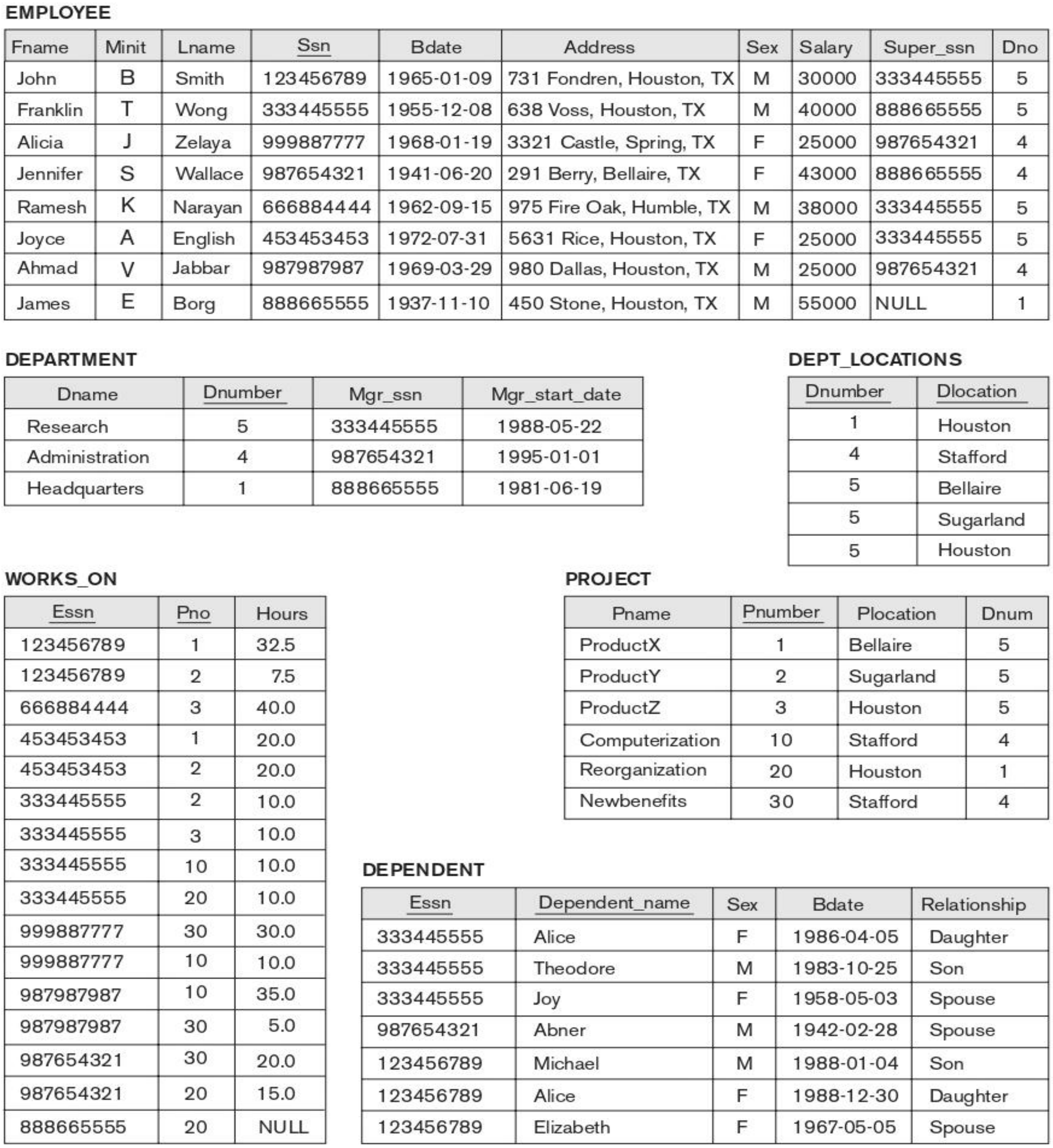
The relationship is described by the foreign key super_ssn of the employee relation.

Ex: Retrieve the details of supervisors.

Examples of Queries in Relational Algebra

All examples refer to the database in the following Figure. In general, the same query can be stated in numerous ways using the various operations.

Figure 5.6
One possible database state for the COMPANY relational database schema.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

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

DEPT_LOCATIONS

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

WORKS_ON

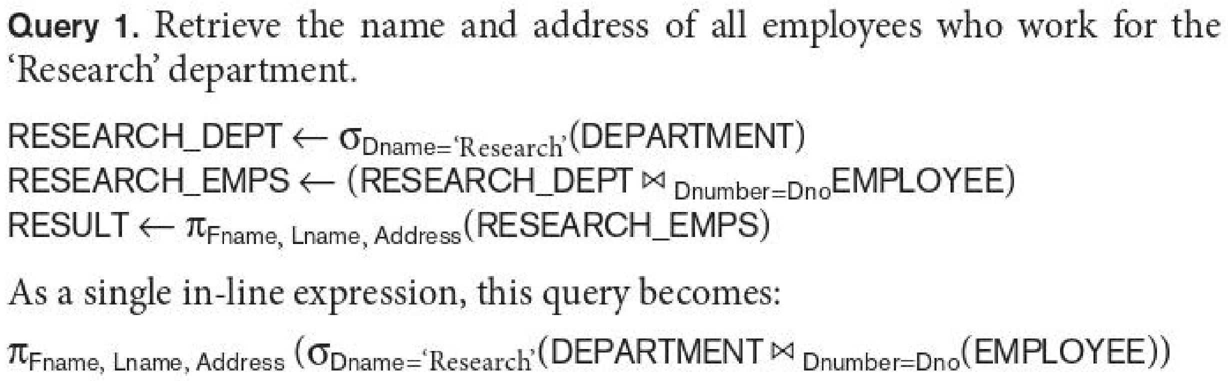
Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

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

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

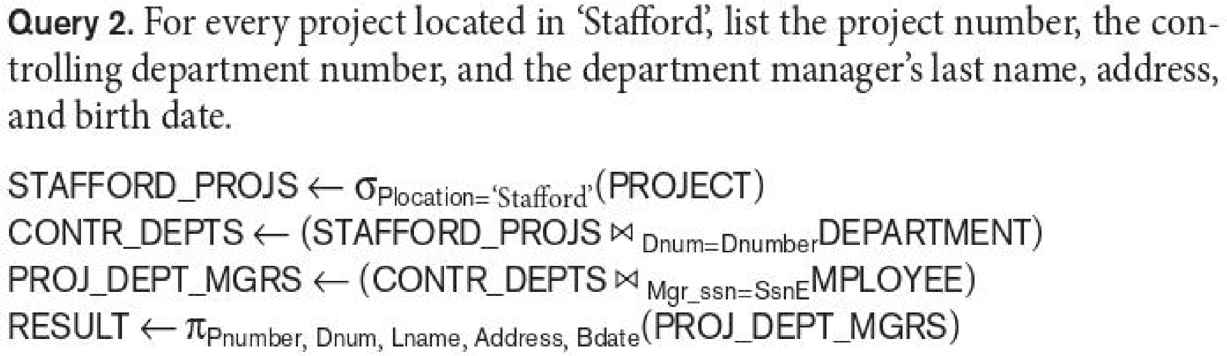


Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{Dname='Research'}}(\text{DEPARTMENT})$
$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$
$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Address}}(\text{RESEARCH\_EMPS})$

As a single in-line expression, this query becomes:

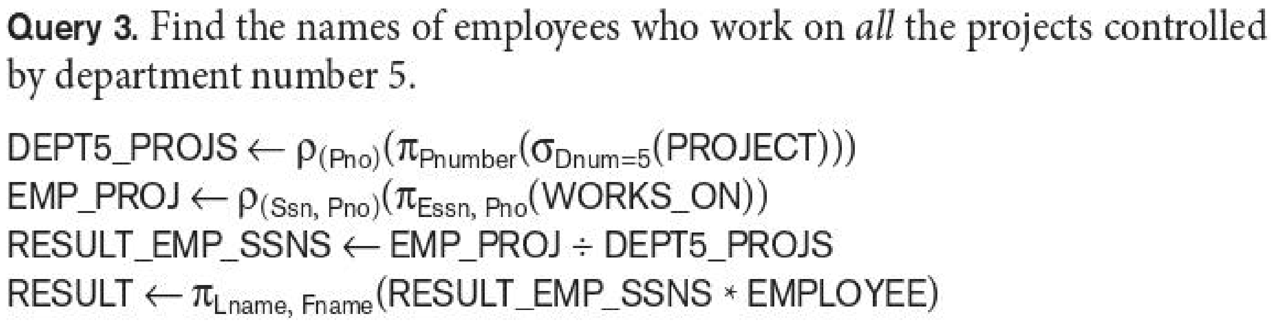
$\pi_{\text{Fname, Lname, Address}} (\sigma_{\text{Dname='Research'}}(\text{DEPARTMENT} \bowtie_{\text{Dnumber=Dno}}(\text{EMPLOYEE}))$



Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$
$\text{CONTR\_DEPTS} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT})$
$\text{PROJ\_DEPT\_MGRS} \leftarrow (\text{CONTR\_DEPTS} \bowtie_{\text{Mgr\_ssn=SsnE}} \text{MPLOYEE})$
$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGRS})$

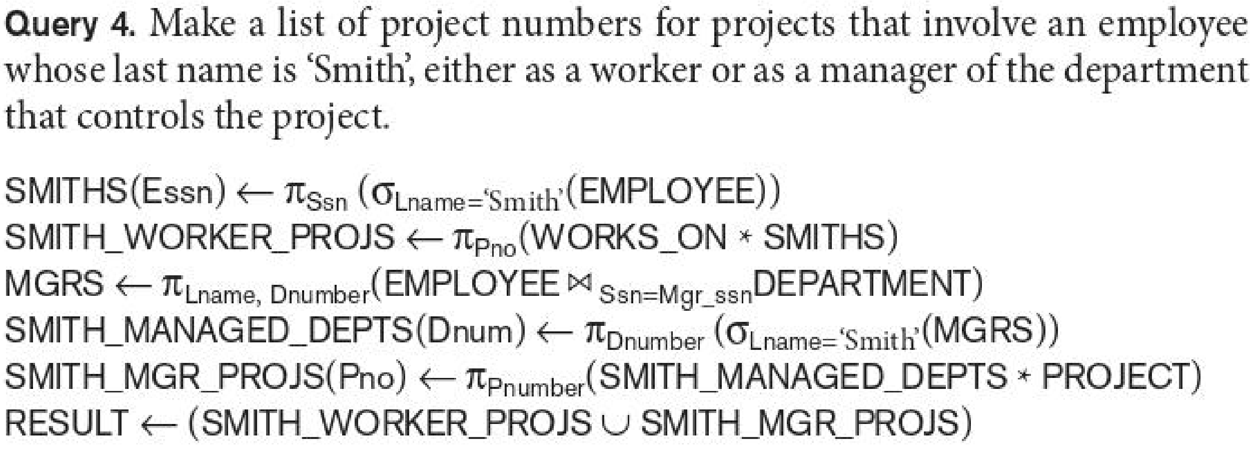
In this example, we first select the projects located in Stafford, then join them with their controlling departments, and then join the result with the department managers. Finally, we apply a project operation on the desired attributes



Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

$\text{DEPT5\_PROJS} \leftarrow \rho_{(\text{Pno})}(\pi_{\text{Pnumber}}(\sigma_{\text{Dnum=5}}(\text{PROJECT})))$
$\text{EMP\_PROJ} \leftarrow \rho_{(\text{Ssn, Pno})}(\pi_{\text{Essn, Pno}}(\text{WORKS\_ON}))$
$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$
$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$

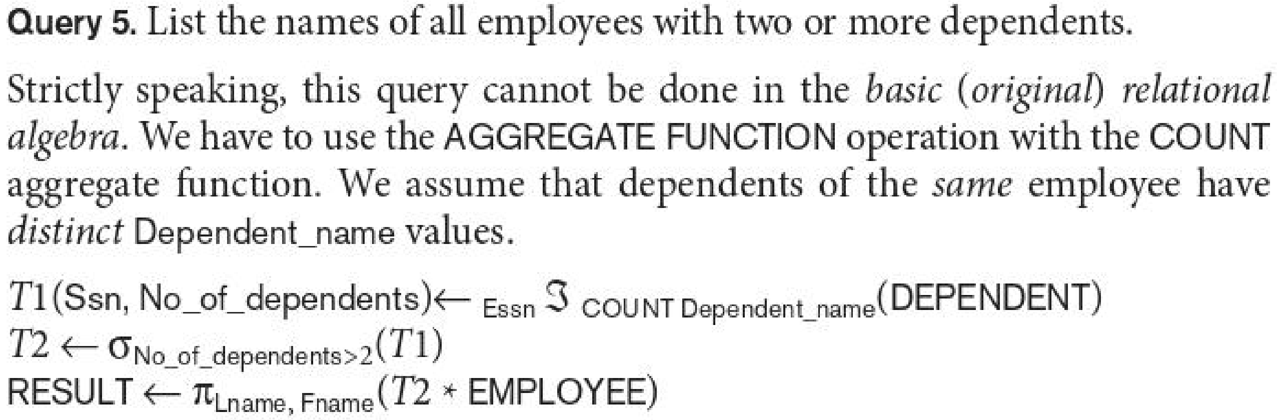
In this query, we first create a table DEPT5_PROJS that contains the project numbers of all projects controlled by department 5. Then we create a table EMP_PROJ that holds (Ssn, Pno) tuples, and apply the division operation. Finally, we join the result of the division, which holds only Ssn values, with the EMPLOYEE table to retrieve the Fname, Lname attributes from EMPLOYEE.



Query 4. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

SMITHS(Essn) ← π_{Ssn} ($\sigma_{\text{Lname='Smith'}}$(EMPLOYEE))
SMITH_WORKER_PROJS ← π_{Pno}(WORKS_ON * SMITHS)
MGRS ← $\pi_{\text{Lname, Dnumber}}$(EMPLOYEE $\bowtie_{\text{Ssn=Mgr_ssn}}$DEPARTMENT)
SMITH_MANAGED_DEPTS(Dnum) ← π_{Dnumber} ($\sigma_{\text{Lname='Smith'}}$(MGRS))
SMITH_MGR_PROJS(Pno) ← π_{Pnumber}(SMITH_MANAGED_DEPTS * PROJECT)
RESULT ← (SMITH_WORKER_PROJS $\cup$ SMITH_MGR_PROJS)

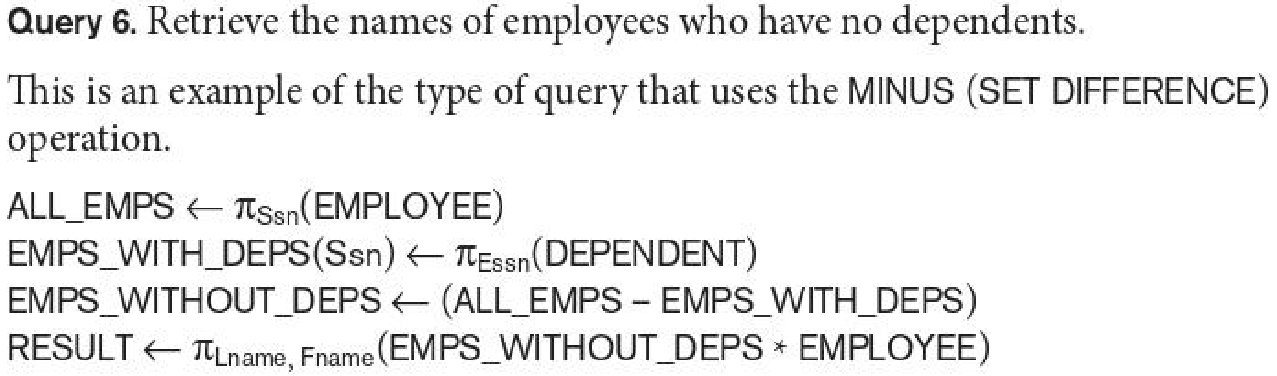
In this query, we retrieved the project numbers for projects that involve an employee named Smith as a worker in SMITH_WORKER_PROJS. Then we retrieved the project numbers for projects that involve an employee named Smith as manager of the department that controls the project in SMITH_MGR_PROJS. Finally, we applied the UNION operation on SMITH_WORKER_PROJS and SMITH_MGR_PROJS.



Query 5. List the names of all employees with two or more dependents.

Strictly speaking, this query cannot be done in the *basic* (*original*) *relational algebra*. We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* Dependent_name values.

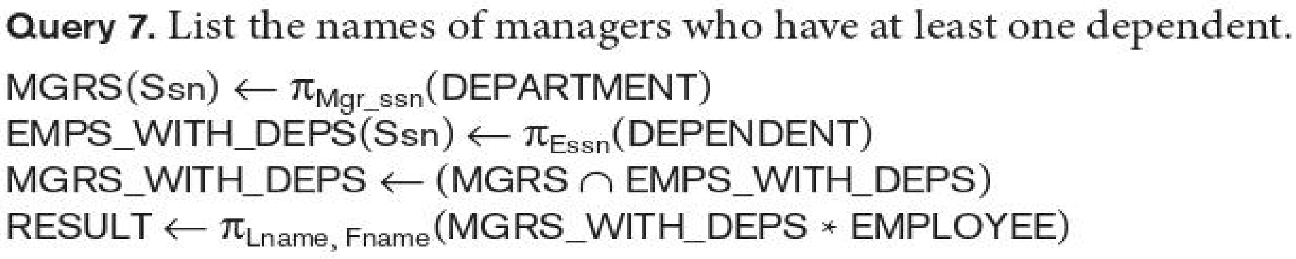
$T1$(Ssn, No_of_dependents)← $_{\text{Essn}}\,\Im\,_{\text{COUNT Dependent\_name}}$(DEPENDENT)
$T2$ ← $\sigma_{\text{No\_of\_dependents>2}}$($T1$)
RESULT ← $\pi_{\text{Lname, Fname}}$($T2$ * EMPLOYEE)



Query 6. Retrieve the names of employees who have no dependents.

This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

ALL_EMPS ← π_{Ssn}(EMPLOYEE)
EMPS_WITH_DEPS(Ssn) ← π_{Essn}(DEPENDENT)
EMPS_WITHOUT_DEPS ← (ALL_EMPS – EMPS_WITH_DEPS)
RESULT ← $\pi_{\text{Lname, Fname}}$(EMPS_WITHOUT_DEPS * EMPLOYEE)



Query 7. List the names of managers who have at least one dependent.

$\text{MGRS(Ssn)} \leftarrow \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$

$\text{EMPS\_WITH\_DEPS(Ssn)} \leftarrow \pi_{\text{Essn}}(\text{DEPENDENT})$

$\text{MGRS\_WITH\_DEPS} \leftarrow (\text{MGRS} \cap \text{EMPS\_WITH\_DEPS})$

$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{MGRS\_WITH\_DEPS} * \text{EMPLOYEE})$

In this query, we retrieve the Ssns of managers in MGRS, and the Ssns of employees with at least one dependent in EMPS_WITH_DEPS, then we apply the SET INTERSECTION operation to get the Ssns of managers who have at least one dependent.

RELATIONAL CALCULUS

We introduce another formal query language for the relational model called relational calculus. Relational algebra is procedural query language. In that we must write a sequence of operations in a particular order. Relational calculus is a nonprocedural query language. A relational calculus expression specifies what is to be retrieved rather than how to retrieve.

Tuple Relational Calculus:

The tuple relational calculus is based on specifying a number of tuple variables. Each tuple variable usually renges over a particular database relation, meaning that the variable may take as its value any individual tuple from that relation.

A simple tupe relational calculus query is of the form:

{ t | COND(t) }

Where t is a tuple variable and COND(t) is a conditional (Boolean) expression. The result of such query is the set of all tuples t that evaluate COND(t) to TRUE. These tuples are said to satisfy COND(t).

For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

{ t | EMPLOYEE(t) AND t.Salary>50000}

The condition EMPLOYEE(t) specifies that the range relation of tuple variable t is EMPLOYEE. Each EMPLOYEE tuple t that satisfies the condition t.Salary>50000 will be retrieved.

The above query retrieves all attribute values for each selected EMPLOYEE tuple t. To retrieve only some of the attributes—say, the first and last names—we write

{t.Fname, t.Lname | EMPLOYEE(t) AND t.Salary>50000}

Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply domain calculus. Historically, while SQL, which was based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language called QBE (Query-By-Example), which is related to domain calculus, was being developed at the IBM T. J. Watson Research Center in Yorktown Heights, New York.

Domain calculus differs from tuple calculus in the type of variables used in formulas:

Rather than having variables range over tuples, the variables range over single values from domains of attributes.

To form a relation of degree n for a query result, we must have n of these domain variables—one for each attribute. An expression of the domain calculus is of the form

{x1, x2, ..., xn | COND(x1, x2, ..., xn, xn+1, xn+2, ..., xn+m)}

where x1, x2, … , xn, xn+1, xn+2, … , xn+m are domain variables that range over domains (of attributes), and COND is a condition or formula of the domain relational calculus.

A formula is made up of atoms. The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form R(x1, x2, … , xj), where R is the name of a relation of degree j and each xi, 1 ≤ i ≤ j, is a domain variable. This atom states that a list of values of <x1,x2,…xj> must be a tuple in the relation whose name is R, where xi is the value of the ith attribute value of the tuple. To make a domain calculus expression more concise, we can drop the commas in a list of variables; thus, we can write:

   {x1, x2, ..., xn | R(x1 x2 x3) AND ...} instead of:
   {x1, x2, ... , xn | R(x1, x2, x3) AND ...}

2. An atom of the form **xi op xj** , where **op** is one of the comparison operators in the set {=,<,>,≤ , ≥, ≠}, and xi and xj are domain variables.

3. An atom of the form **xi op c** or **c op xj** , where op is one of the comparison operators in the set {=,<,>,≤ , ≥, ≠}, xi and xj are domain variables, and c is a constant value.

In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation R, then the atom is TRUE.

In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers.

We will use lowercase letters l, m, n, … , x, y, z for domain variables.

Query 0: List the birth date and address of the employee whose name is ‘John B. Smith’.

Q0: {u, v | (∃q) (∃r) (∃s) (∃t) (∃w) (∃x) (∃y) (∃z)

(EMPLOYEE(qrstuvwxyz) AND q=‘John’ AND r=‘B’ AND s=‘Smith’)}

We need ten variables for the EMPLOYEE relation, one to range over each of the domains of attributes of EMPLOYEE in order. Of the ten variables q, r, s, … , z, only u and v are free, because they appear to the left of the bar and hence should not be bound to a quantifier.

We first specify the requested attributes, Bdate and Address, by the free domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (|)—namely, that the sequence of values assigned to the variables qrstuvwxyz be a tuple of the EMPLOYEE relation and that the values for q (Fname), r (Minit), and s (Lname) be equal to 'John', 'B', and 'Smith', respectively.

Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

Q1: {q, s, v | (∃z) (∃l) (∃m) (EMPLOYEE(qrstuvwxyz) AND DEPARTMENT(lmno) AND l='Research' AND m=z)}

A condition relating two domain variables that range over attributes from two relations, such as m = z in Q1, is a join condition, whereas a condition that relates a domain variable to a constant, such as l = 'Research', is a selection condition.

Query 2: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

Q2: {i, k, s, u, v | (∃j)(∃m)(∃n)(∃t)(PROJECT(hijk) AND EMPLOYEE(qrstuvwxyz) AND DEPARTMENT(lmno) AND k=m AND n=t AND j='Stafford')}

Query 6: List the names of employees who have no dependents.

Q6: {q, s | (∃t)(EMPLOYEE(qrstuvwxyz) AND (NOT(∃l)(DEPENDENT(lmnop) AND t=l)))}

Query 7: List the names of managers who have at least one dependent.

Q7: {s, q | (∃t)(∃j)(∃l)(EMPLOYEE(qrstuvwxyz) AND DEPARTMENT(hijk) AND DEPENDENT(lmnop) AND t=j AND l=t)}

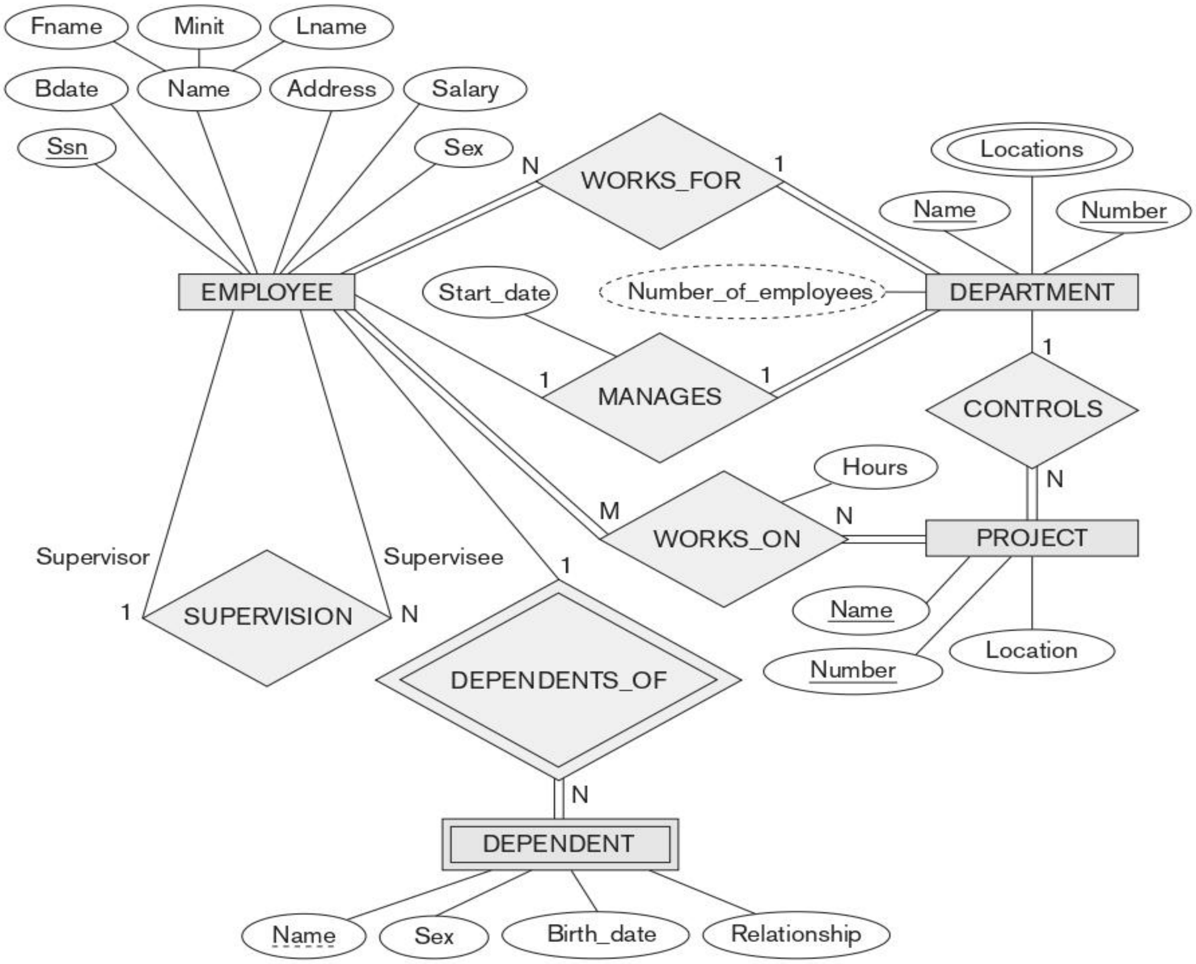
Relational Database Design by ER- and EER-to-Relational Mapping

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.

An ER Schema diagram for the COMPANY database.



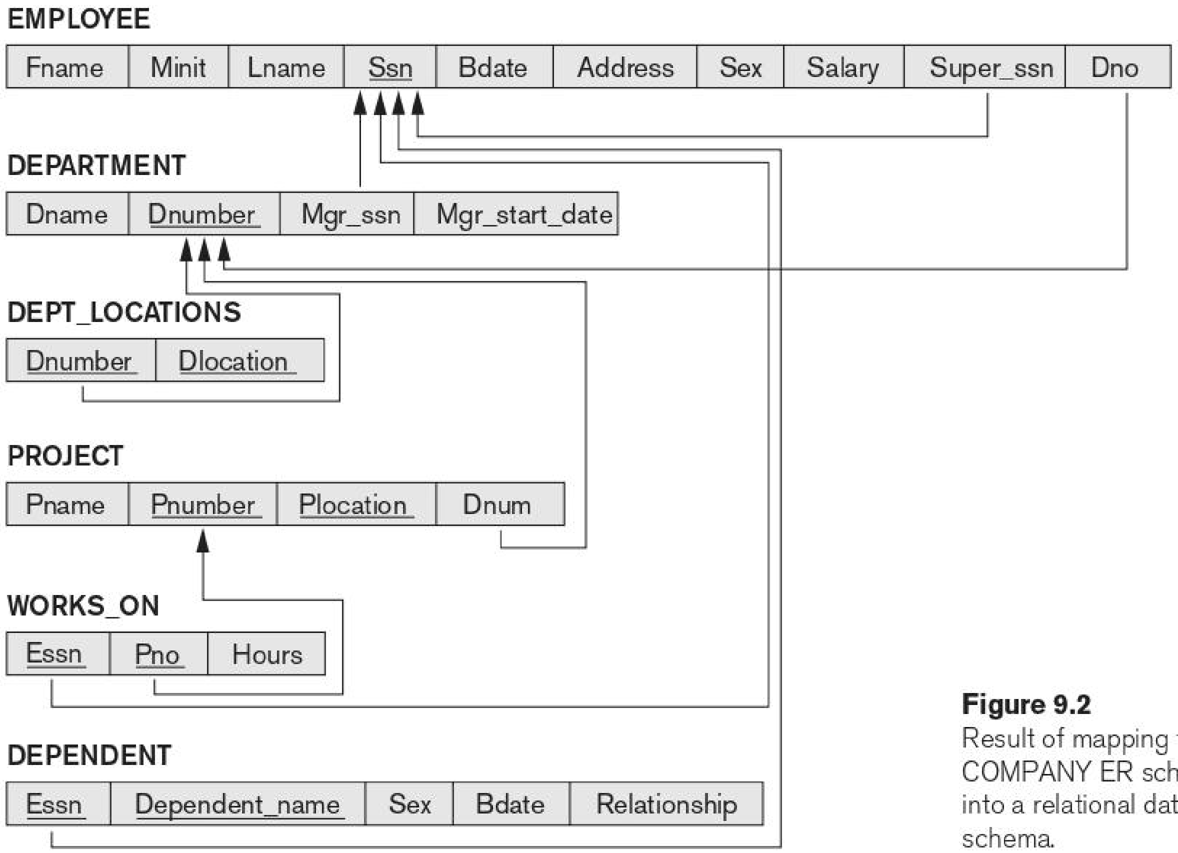


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

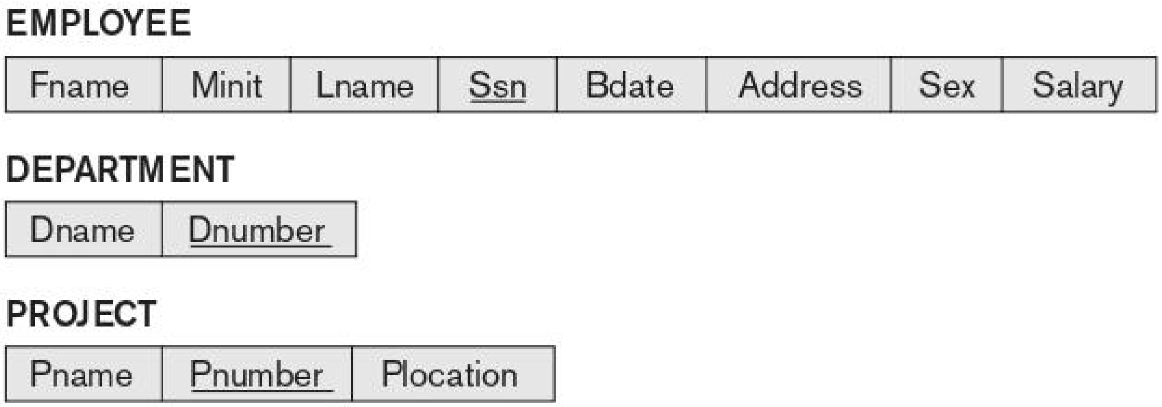
ER to Relational Mapping Algorithms:

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.

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

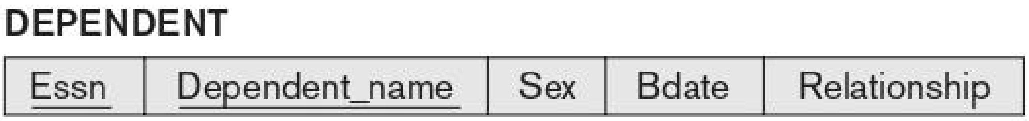
PROJECT

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'.

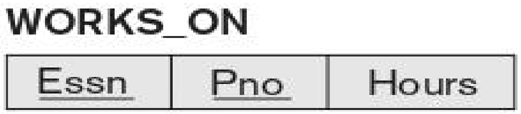
In our example, we map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION.

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.**



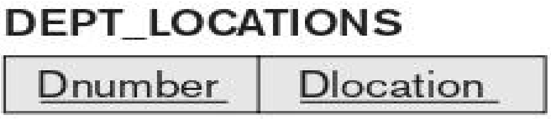
WORKS_ON

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}.**



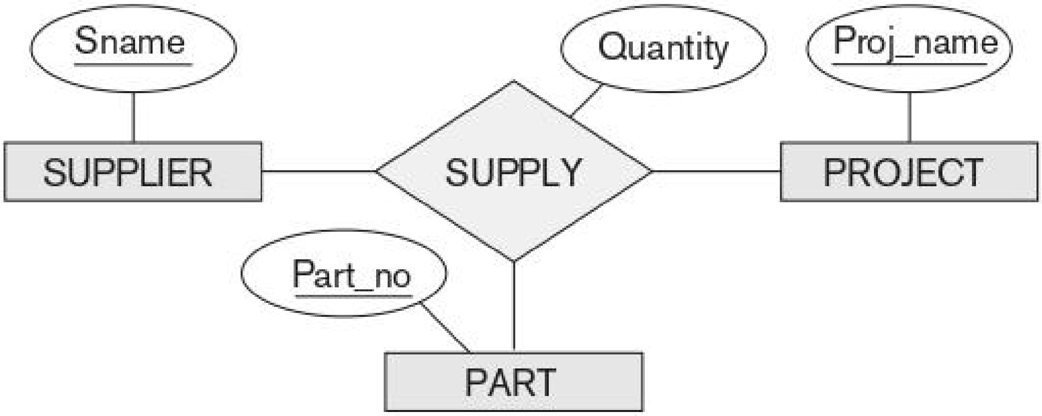
DEPT_LOCATIONS

Dnumber	Dlocation

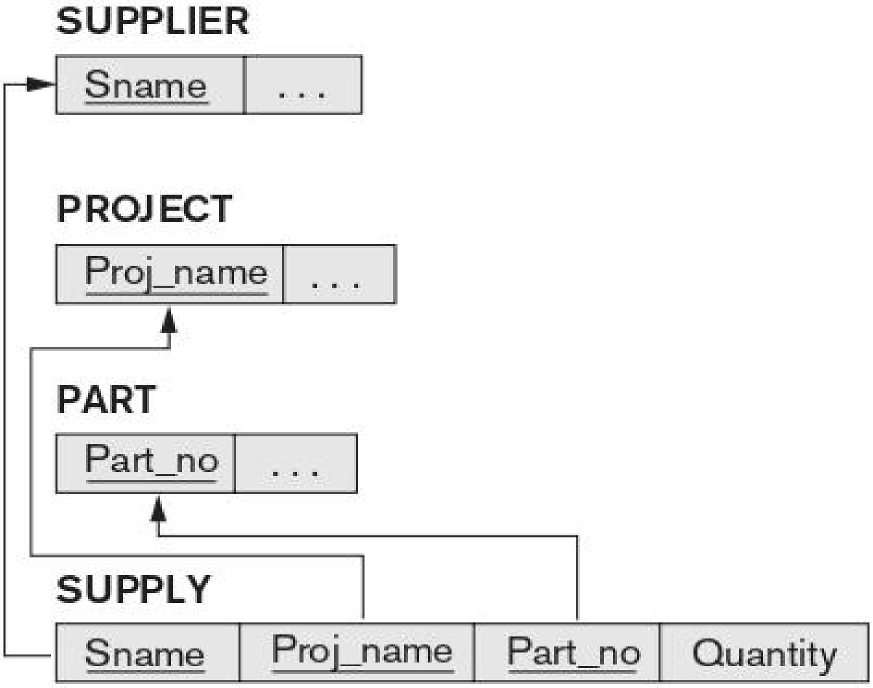
Step 7: Mapping of N-ary Relationship Types:

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.



Mapping EER Model Constructs to Relations

The mapping of EER model constructs to relations by extending the ER-to-relational mapping algorithm.

Mapping of Specialization or Generalization

There are several options for mapping a number of subclasses that together form a specialization such as the {SECRETARY, TECHNICIAN, ENGINEER} subclasses of EMPLOYEE. The two main options are to map the whole specialization into a **single table**, or to map it into **multiple tables**. Within each option are variations that depend on the constraints on the specialization/generalization.

Step 8: Options for Mapping Specialization or Generalization:

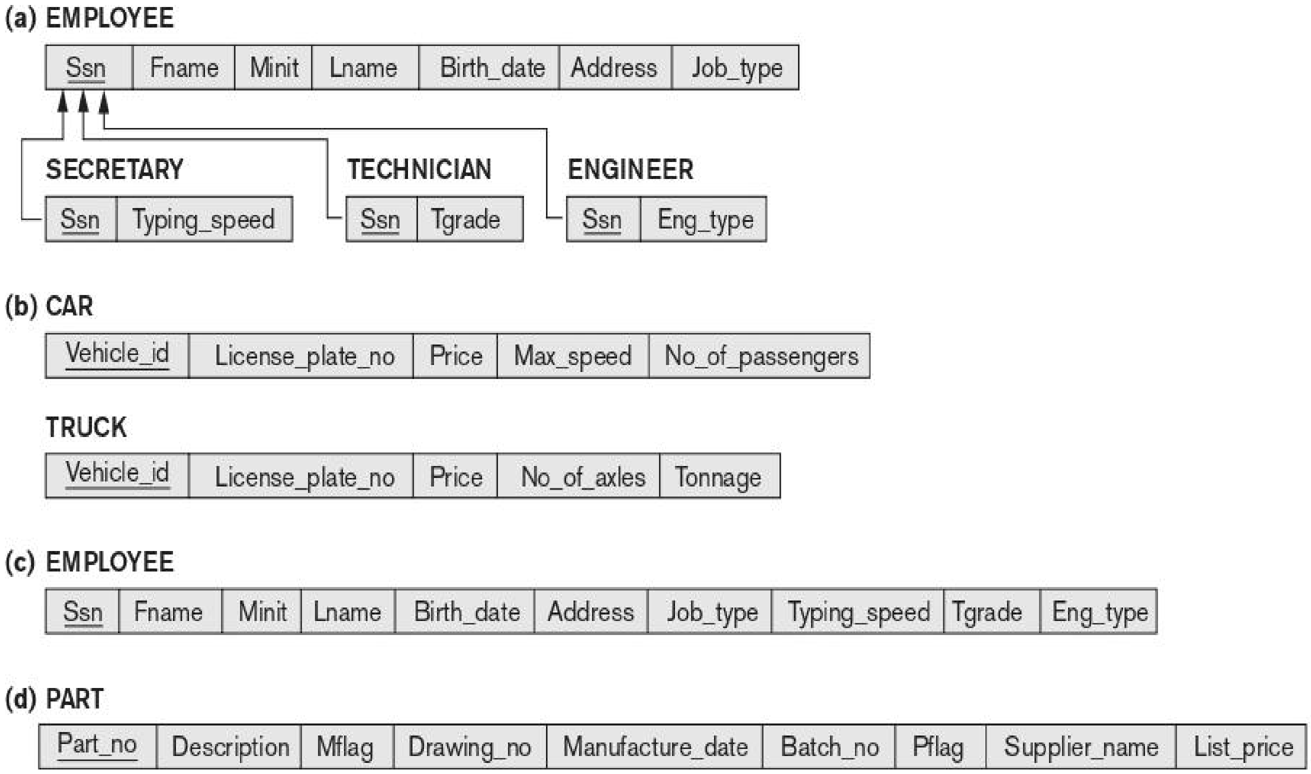
Convert each specialization with m subclasses {S1, S2, … , Sm} and (generalized) superclass C, where the attributes of C are {k, a1, … , an} and k is the (primary) key, into relation schemas using one of the following options:

Option 8A: Multiple relations—superclass and subclasses: Create a relation L for C with attributes Attrs(L) = {k, a1, … , an} and PK(L) = k. Create a relation Li for each subclass Si, 1 ≤ i ≤ m, with the attributes Attrs(Li) = {k} ∪ {attributes of Si} and PK(Li) = k.

Option 8B: Multiple relations—subclass relations only: Create a relation Li for each subclass Si, 1 ≤ i ≤ m, with the attributes Attrs(Li) = {attributes of Si} ∪ {k, a1, … , an} and PK(Li) = k. This option only works for a specialization whose subclasses are total (every entity in the superclass must belong to (at least) one of the subclasses).

Option 8C: Single relation with one type attribute: Create a single relation L with attributes Attrs(L) = {k, a1, …, an} ∪ {attributes of S1} ∪ … ∪ {attributes of Sm} ∪ {t} and PK(L) = k. The attribute t is called a type (or discriminating) attribute whose value indicates the subclass to which each tuple belongs, if any. This option works only for a specialization whose subclasses are disjoint, and has the potential for generating many NULL values if many specific (local) attributes exist in the subclasses.

Option 8D: Single relation with multiple type attributes. Create a single relation schema L with attributes Attrs(L) = {k, a1, …, an} ∪ {attributes of S1} ∪ … ∪ {attributes of Sm} ∪ {t1, t2, …, tm} and PK(L) = k. Each ti, 1 ≤ i ≤ m, is a Boolean type attribute indicating whether or not a tuple belongs to subclass Si. This option is used for a specialization whose subclasses are overlapping.



(a) EMPLOYEE

Ssn	Fname	Minit	Lname	Birth_date	Address	Job_type

SECRETARY

Ssn	Typing_speed

TECHNICIAN

Ssn	Tgrade

ENGINEER

Ssn	Eng_type

(b) CAR

Vehicle_id	License_plate_no	Price	Max_speed	No_of_passengers

TRUCK

Vehicle_id	License_plate_no	Price	No_of_axles	Tonnage

(c) EMPLOYEE

Ssn	Fname	Minit	Lname	Birth_date	Address	Job_type	Typing_speed	Tgrade	Eng_type

(d) PART

Part_no	Description	Mflag	Drawing_no	Manufacture_date	Batch_no	Pflag	Supplier_name	List_price

Figure 9.5

Options for mapping specialization or generalization. (a) Mapping the EER schema in Figure 4.4 using option 8A. (b) Mapping the EER schema in Figure 4.3(b) using option 8B. (c) Mapping the EER schema in Figure 4.4 using option 8C. (d) Mapping Figure 4.5 using option 8D with Boolean type fields Mflag and Pflag.

Mapping of Shared Subclasses (Multiple Inheritance):

A shared subclass, such as ENGINEERING_MANAGER is a subclass of several superclasses, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category (union type). We can apply any of the options in step 8 to a shared subclass, subject to the restrictions in step 8 of the mapping algorithm.

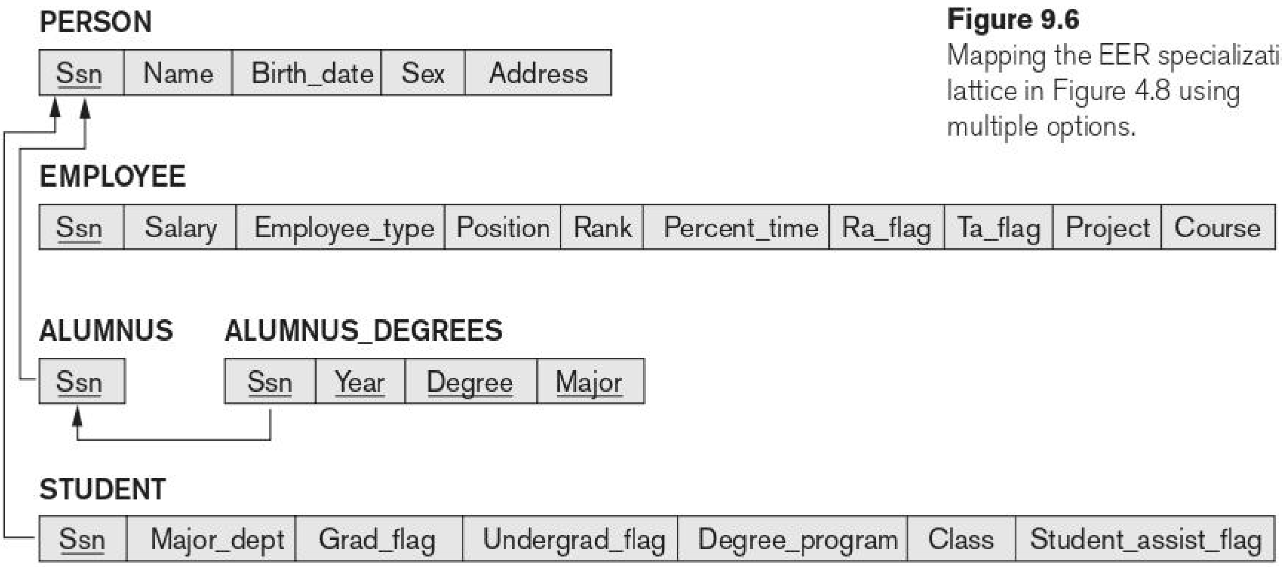

PERSON
Ssn | Name | Birth_date | Sex | Address
EMPLOYEE
Ssn | Salary | Employee_type | Position | Rank | Percent_time | Ra_flag | Ta_flag | Project | Course
ALUMNUS
Ssn
ALUMNUS_DEGREES
Ssn | Year | Degree | Major
STUDENT
Ssn | Major_dept | Grad_flag | Undergrad_flag | Degree_program | Class | Student_assist_flag

Figure 9.6
Mapping the EER specialization lattice in Figure 4.8 using multiple options.

In the above Figure, options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT. Option 8C is used in the EMPLOYEE relation (Employee_type attribute) and option 8D is used in the STUDENT relation (Student_assist_flag attribute).

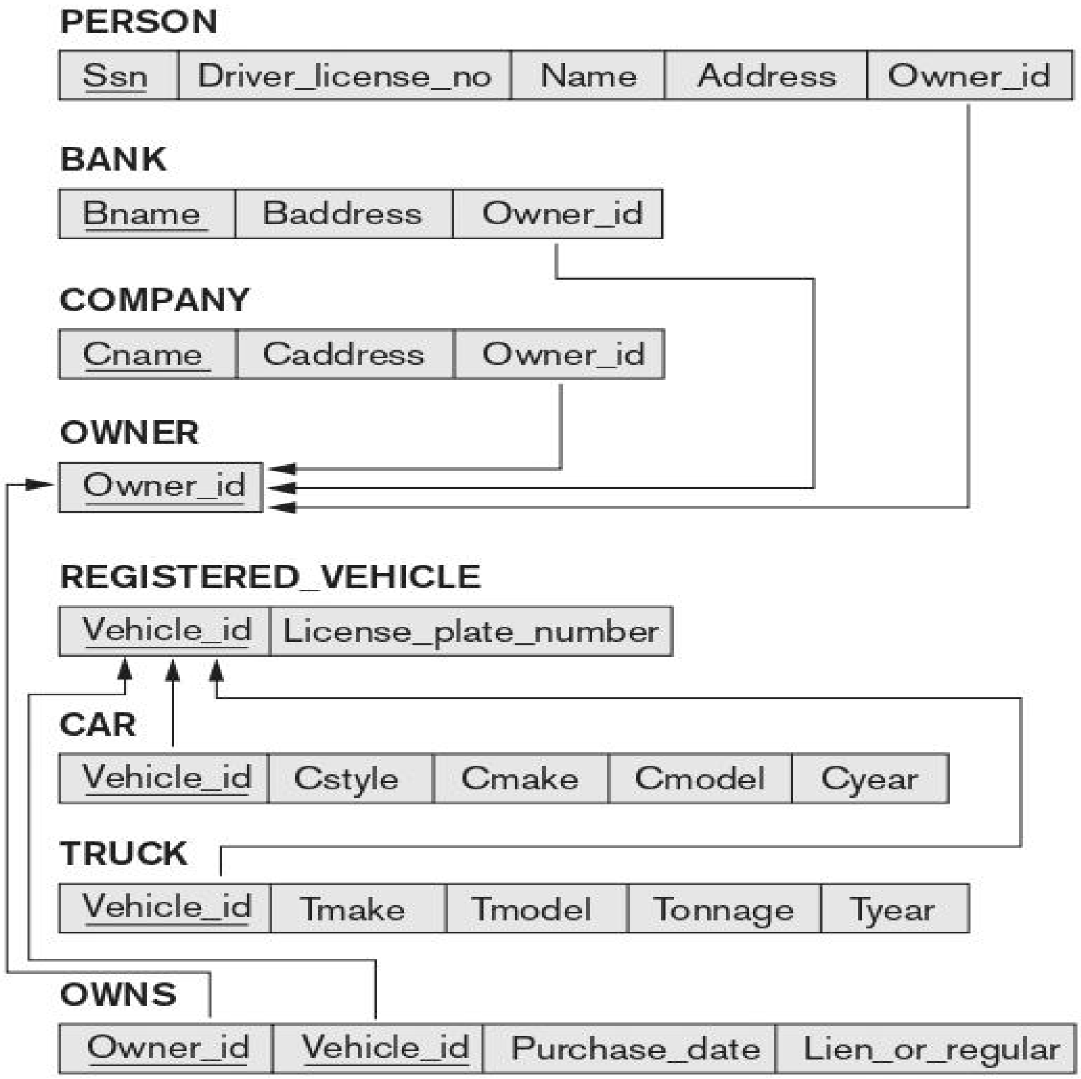
Mapping of Categories (Union Types)

We add another step to the mapping procedure—step 9—to handle categories. A category (or union type) is a subclass of the union of two or more superclasses that can have different keys. An example is the OWNER category, which is a subset of the union of three entity types PERSON, BANK, and COMPANY. The other category in that REGISTERED_VEHICLE, has two superclasses that have the same key attribute.

Step 9: Mapping of Union Types (Categories): For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a surrogate key, when creating a relation to correspond to the union type. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the relation.

In our example, we create a relation OWNER to correspond to the OWNER category, as illustrated in the following Figure, and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called Owner_id.

Figure: Mapping the EER categories (union types)



Schema Definition, Basic Constraints and Queries

SQL Data definition

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), types, and domains, as well as other constructs such as views, assertions, and triggers.

Schema and Catalog Concepts in SQL

An **SQL schema** is identified by a schema name and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, types, constraints, views, domains, and other constructs.

A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions.

For example, the following statement creates a schema called COMPANY owned by the user with authorization identifier 'Jsmith'. Note that each statement in SQL ends with a semicolon.

CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';

The CREATE TABLE Command in SQL:

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and possibly attribute constraints, such as NOT NULL.

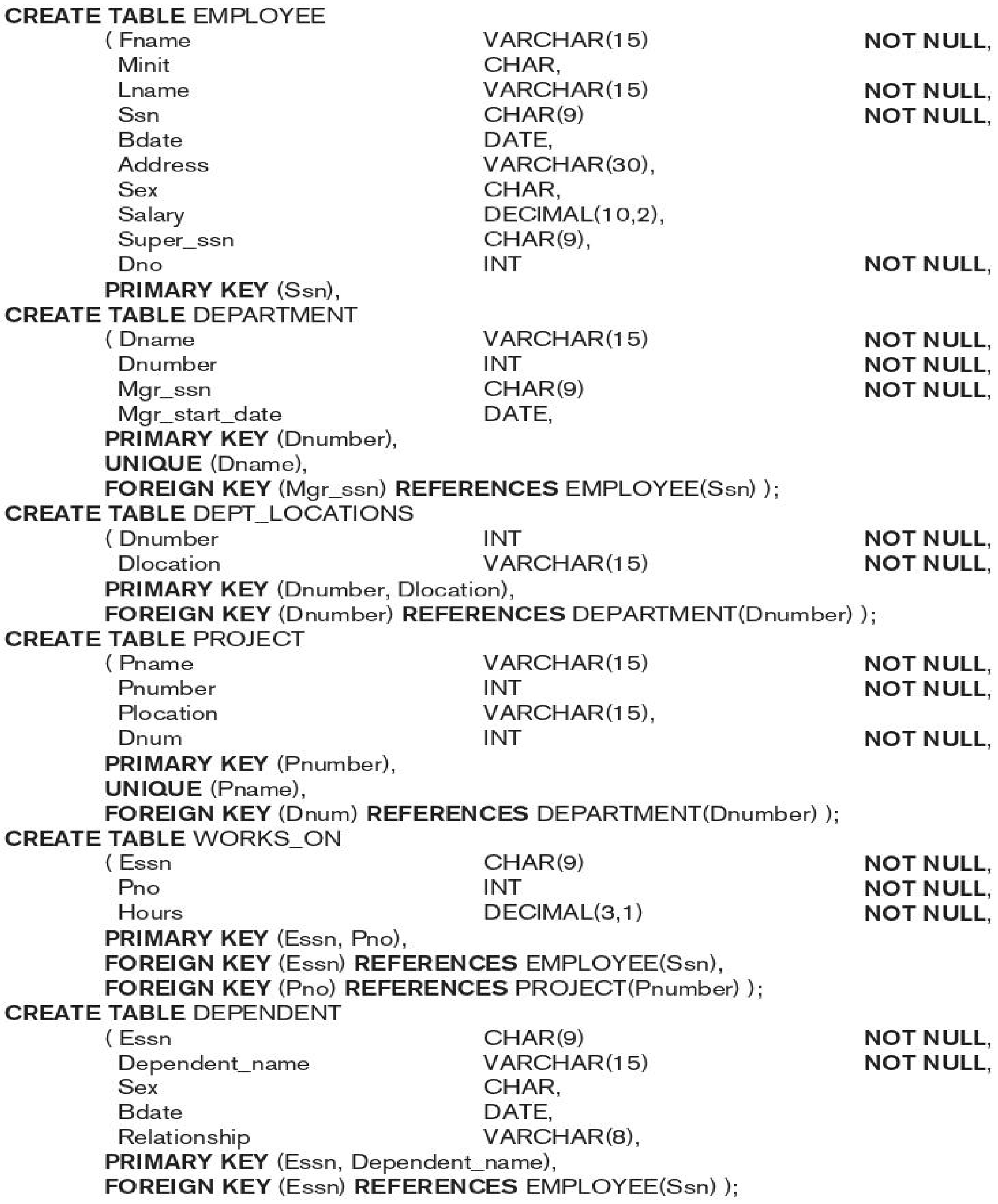
Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

CREATE TABLE COMPANY.EMPLOYEE

rather than

CREATE TABLE EMPLOYEE

SQL CREATE TABLE data definition statements for defining the COMPANY schema:



```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Attribute Data Types:

The basic data types available for attributes include numeric, character string, bit string, Boolean, date, and time.

Numeric data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION).

Character-string data types are either fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters—or varying length—VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters.

A Boolean data type has the traditional values of TRUE or FALSE.

The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by the SQL implementation.

Specifying Constraints in SQL

Specifying Attribute Constraints and Attribute Defaults:

Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation.

It is also possible to define a default value for an attribute by appending the clause DEFAULT to an attribute definition.

CREATE TABLE DEPARTMENT (… , Mgr_ssn CHAR(9) NOT NULL DEFAULT ‘888665555’, … ,)

Specifying Key and Referential Integrity Constraints:

Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them.

The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly. For example, the primary key of DEPARTMENT can be specified as follows:

Dnumber INT PRIMARY KEY

The UNIQUE clause specifies alternate (unique) keys, also known as candidate keys as illustrated in the DEPARTMENT and PROJECT table declarations. The UNIQUE clause can also be specified directly for a unique key if it is a single attribute, as in the following example:

Dname VARCHAR(15) UNIQUE,

The FOREIGN KEY clause, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is updated.

CREATE TABLE DEPARTMENT (Dname Dnumber Mgr_ssn Mgr_start_date VARCHAR(15) INT CHAR(9) DATE, NOT NULL, NOT NULL, NOT NULL, **PRIMARY KEY** (Dnumber), **UNIQUE** (Dname), **FOREIGN KEY** (Mgr_ssn) REFERENCES EMPLOYEE(Ssn));

Giving Names to Constraints:

A constraint may be given a constraint name, following the keyword CONSTRAINT. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint.

Schema Change Statements in SQL

The schema evolution commands available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

The DROP Command:

The DROP command can be used to drop named schema elements, such as tables, domains, types, or constraints. One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two drop behavior options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

```
DROP SCHEMA   COMPANY   CASCADE;
```

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

The DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command should be used instead of DROP TABLE.

The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

The ALTER Command:

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema, we can use the command

ALTER TABLE COMPANY.EMPLOYEE **ADD COLUMN** Job VARCHAR(12);

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column. For example, the following command removes the attribute Address from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE;**

One can also change the constraints specified on a table by adding or dropping a named constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK from the EMPLOYEE relation, we write:

```
ALTER TABLE     COMPANY.EMPLOYEE
DROP CONSTRAINT   EMPSUPERFK CASCADE;
```

Basic Queries in SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. The SELECT statement is not the same as the SELECT operation of relational algebra. There are many options and flavors to the SELECT statement in SQL.

The SELECT-FROM-WHERE Structure of Basic SQL Queries

Queries in SQL can be very complex. We will start with simple queries, and then progress to more complex ones in a step-by-step manner.

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
 SELECT   <attribute list>
 FROM    <table list>
WHERE    <condition>;
```

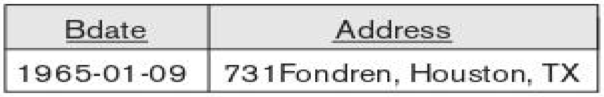
Where

<attribute list>   is a list of attribute names whose values are to be retrieved by the query.
<table list>   is a list of the relation names required to process the query.
<condition>   is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <=, >, >=, and <>. These correspond to the relational algebra operators =, , ≥, and ≠, respectively, and to the C/C++ programming language operators =, <=, >, >=, and !=. The main syntactic difference is the **not equal** operator.

Query 0: Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:    SELECT  Bdate, Address
       FROM   EMPLOYEE
      WHERE   Fname = 'John' AND Minit = 'B' AND Lname = 'Smith';
```

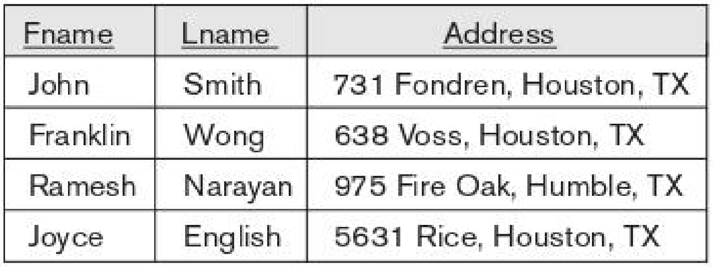


Bdate	Address
1965-01-09	731Fondren, Houston, TX

This query involves only the EMPLOYEE relation listed in the FROM clause. The query selects the individual EMPLOYEE tuples that satisfy the condition of the WHERE clause, then projects the result on the Bdate and Address attributes listed in the SELECT clause.

Query 1: Retrieve the name and address of all employees who work for the ‘Research’ department.

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
     WHERE     Dname = ‘Research’ AND Dnumber = Dno;
```



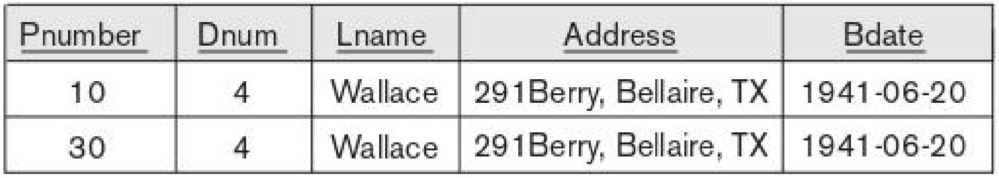
Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

In the WHERE clause of Q1, the condition Dname = ‘Research’ is a selection condition that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a join condition, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE.

A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join** query. The next example is a select-project-join query with two join conditions.

Query 2: For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
     WHERE     Dnum = Dnumber AND Mgr_ssn = Ssn AND Plocation =
‘Stafford’;
```



Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn = Ssn relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a combination of one project, one department and one employee. The projection attributes are used to choose the attributes to be displayed from each combined tuple.

Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables:

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different tables. If this is the case, and a multitable query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity.

Query Q1 would be rephrased as shown in Q1A. We must prefix the attributes Name and Dnumber in Q1A to specify which ones we are referring to, because the same attribute names are used in both relations:

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
       FROM    EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name = ‘Research’ AND
               DEPARTMENT.Dnumber = EMPLOYEE.Dnumber;
```

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice, as in the following example.

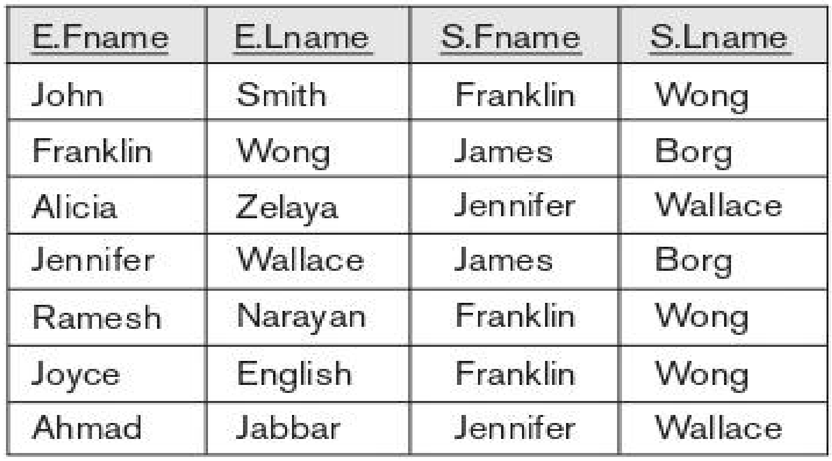
Query 8: For each employee, retrieve the employee’s first and last name and the first and last name of his or her immediate supervisor.

```
Q8:  SELECT    E.Fname, E.Lname, S.Fname, S.Lname
     FROM      EMPLOYEE AS E, EMPLOYEE AS S
     WHERE     E.Super_ssn = S.Ssn;
```

In this case, we are required to declare alternative relation names E and S, called aliases or tuple variables, for the EMPLOYEE relation. An alias can follow the keyword **AS**, or it can directly follow the relation name—for example, by writing EMPLOYEE **E**, EMPLOYEE **S** in the FROM clause of Q8. It is also possible to rename the relation attributes within the query in SQL by giving them aliases. For example, if we write

EMPLOYEE **AS** E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)

in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on.



E.Fname	E.Lname	S.Fname	S.Lname
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

Unspecified WHERE Clause and Use of the Asterisk:

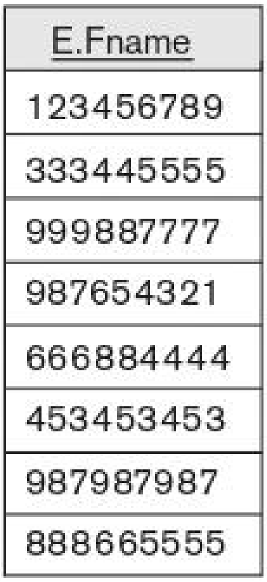
A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected.

For example, Query 9 selects all EMPLOYEE Ssns and Query 10 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not.

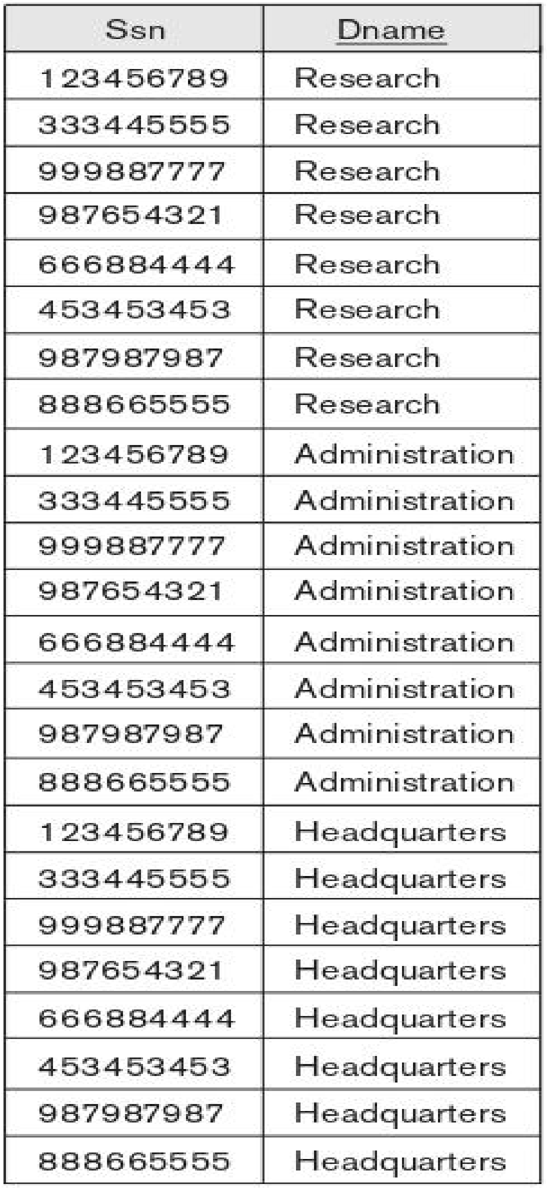
Queries 9 and 10: Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:     SELECT   Ssn
        FROM     EMPLOYEE;
```

```
Q10:    SELECT Ssn, Dname
        FROM EMPLOYEE, DEPARTMENT;
```



E.Fname
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555



Ssn	Dname
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes. The * can also be prefixed by the relation name or alias; for example, EMPLOYEE . * refers to all attributes of the EMPLOYEE table.

Examples:

Q1:
```
SELECT   *
 FROM EMPLOYEE
WHERE Dno = 5;
```

Q2:
```
SELECT  *
FROM EMPLOYEE, DEPARTMENT
WHERE Dname = ‘Research’ AND Dno = Dnumber;
```

Q3:
```
SELECT  *
FROM EMPLOYEE, DEPARTMENT;
```

Tables as Sets in SQL:

SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query.

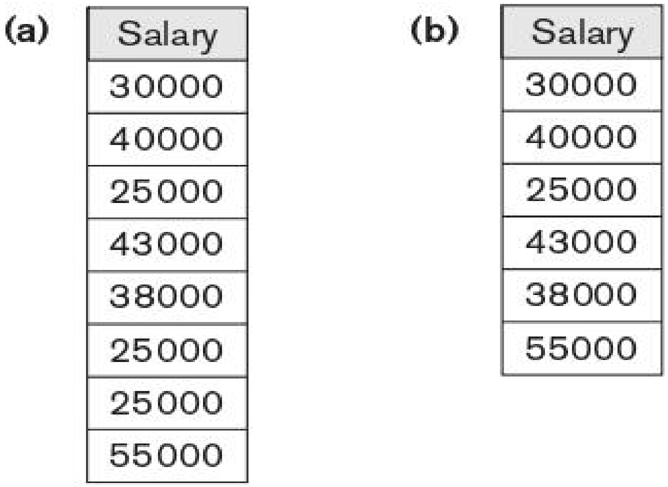
An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple. If we want to eliminate duplicate tuples from the result of an SQL query, we use the keyword DISTINCT in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with **SELECT DISTINCT** eliminates duplicates, whereas a query with **SELECT ALL** does not.

Query 11: Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

Q11:
```
SELECT ALL Salary
FROM EMPLOYEE;
```

Q11A:
```
SELECT DISTINCT Salary
FROM EMPLOYEE;
```

Results of additional SQL queries when applied to the COMPANY database state shown in Figure (a) Q11. (b) Q11A.



(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

SQL has directly incorporated some of the set operations from mathematical set theory, which are also part of relational algebra. There are set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations. The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result.

Query 4: Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4A:  ( SELECT DISTINCT Pnumber
        FROM PROJECT, DEPARTMENT, EMPLOYEE
        WHERE Dnum = Dnumber AND Mgr_ssn = Ssn AND Lname = 'Smith' )
        UNION
       (SELECT DISTINCT Pnumber
        FROM PROJECT, WORKS_ON, EMPLOYEE
       WHERE Pnumber = Pno AND Essn = Ssn AND Lname = 'Smith' );
```

Substring Pattern Matching and Arithmetic Operators:

There are several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query:

Query12: Retrieve all employees whose address is in Houston, Texas.

```
Q12:     SELECT Fname, Lname
         FROM EMPLOYEE
         WHERE Address LIKE ‘%Houston,TX%’;
```

To retrieve all employees who were born during the 1970s, we can use Query Q12A. Here, ‘7’ must be the third character of the string (according to our format for date), so we use the value ‘_ _ 5 _ _ _ _ _ _ _’, with each underscore serving as a placeholder for an arbitrary character.

Query 12: . Find all employees who were born during the 1950s.

```
Q12:      SELECT Fname, Lname
           FROM EMPLOYEE
           WHERE Bdate LIKE ‘_ _ 7 _ _ _ _ _ _ _’;
```

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (−), multiplication (*), and division (/) can be applied to numeric values. For example, suppose all employees who work on the ‘ProductX’ project a 10% raise; we can issue Query 13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

Query 13. Show the resulting salaries if every employee working on the ‘ProductX’ project is given a 10% raise.

```
Q13:   SELECT  E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
       FROM  EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
       WHERE  E.Ssn = W.Essn AND W.Pno = P.Pnumber AND
                 P.Pname = ‘ProductX’;
```

In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator, which can be used for convenience, is BETWEEN, which is illustrated in Query 14.

Query 14: Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
 Q14:   SELECT  *
        FROM  EMPLOYEE
        WHERE  (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

The condition (Salary BETWEEN 30000 AND 40000) in Q14 is equivalent to the condition ((Salary >= 30000) AND (Salary <= 40000)).

Ordering of Query Results:

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the ORDER BY clause.

Query 15: Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
 Q15:  SELECT     D.Dname, E.Lname, E.Fname, P.Pname
       FROM        DEPARTMENT AS D, EMPLOYEE AS E, WORKS_ON AS
W,
                   PROJECT AS P
       WHERE    D.Dnumber = E.Dno AND E.Ssn = W.Essn AND W.Pno =
                   P.Pnumber
      ORDER BY    D.Dname, E.Lname, E.Fname
```

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly.

INSERT, DELETE, and UPDATE Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE.

The INSERT Command:

In its simplest form, INSERT is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.
For example, to add a new tuple to the EMPLOYEE relation and specified in the CREATE TABLE EMPLOYEE … command.

```
Q1:    INSERT INTO EMPLOYEE
       VALUES ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak
                 Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes.

```
Q2:     INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn)
        VALUES ('Richard', 'Marini', 4, '653298653');
```

The DELETE Command:

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command.

```
Q3:    DELETE FROM EMPLOYEE
       WHERE Lname = 'Brown';

Q4:    DELETE FROM EMPLOYEE
       WHERE Ssn = '123456789';

Q5:    DELETE FROM EMPLOYEE
       WHERE Dno = 5;

Q6:    DELETE FROM EMPLOYEE;
```

The UPDATE Command:

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL.

An additional **SET** clause in the UPDATE command specifies the attributes to be modified and their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

```
Q7:      UPDATE   PROJECT
         SET        Plocation = 'Bellaire', Dnum = 5
         WHERE    Pnumber = 10;
```

In the SET clause, the reference to the Salary attribute on the right refers to the old Salary value before modification, and the one on the left refers to the new Salary value after modification:

```
Q8:    UPDATE    EMPLOYEE
       SET       Salary = Salary * 1.1
       WHERE     Dno = 5;
```

More Complex SQL Queries

Nested Queries, Tuples, and Set/Multiset Comparisons:

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries,** which are complete select-from-where blocks within another SQL query. That other query is called the **outer query**. These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed.

In the example, the first nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as manager, whereas the second nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

```
Q:   SELECT DISTINCT Pnumber
      FROM    PROJECT
      WHERE Pnumber IN
                  (SELECT  Pnumber
                   FROM  PROJECT, DEPARTMENT, EMPLOYEE
                   WHERE   Dnum = Dnumber AND Mgr_ssn = Ssn AND Lname
=
                            ‘Smith’ )
                   OR
                   Pnumber  IN
                  (SELECT Pno
                   FROM WORKS_ON, EMPLOYEE
                   WHERE Essn = Ssn AND Lname = ‘Smith’ );
```

SQL allows the use of tuples of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours)    IN  (SELECT Pno, Hours
                           FROM WORKS_ON
                           WHERE Essn = '123456789' );
```

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on. In this example, the **IN** operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL   (SELECT Salary
                      FROM EMPLOYEE
                      WHERE Dno = 5 );
```

Correlated Nested Queries:

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can always be expressed as a single block query.

```
Q:    SELECT   E.Fname, E.Lname
      FROM     EMPLOYEE AS E, DEPENDENT AS D
      WHERE    E.Ssn = D.Essn AND E.Sex = D.Sex
               AND E.Fname = D.Dependent_name;
```

Aggregate Functions in SQL:

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. Grouping is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications, and we will introduce their use in SQL through examples. A number of built-in aggregate functions exist: COUNT, SUM, MAX, MIN, and AVG.

Query: Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q:    SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
      FROM EMPLOYEE;
```

This query returns a single-row summary of all the rows in the EMPLOYEE table.

Query:. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q:  SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
    FROM (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
   WHERE Dname = 'Research';
```

Queries 1 and 2. Retrieve the total number of employees in the company (Q1) and the number of employees in the 'Research' department (Q2).

```
Q1:    SELECT COUNT (*)
       FROM EMPLOYEE;
```

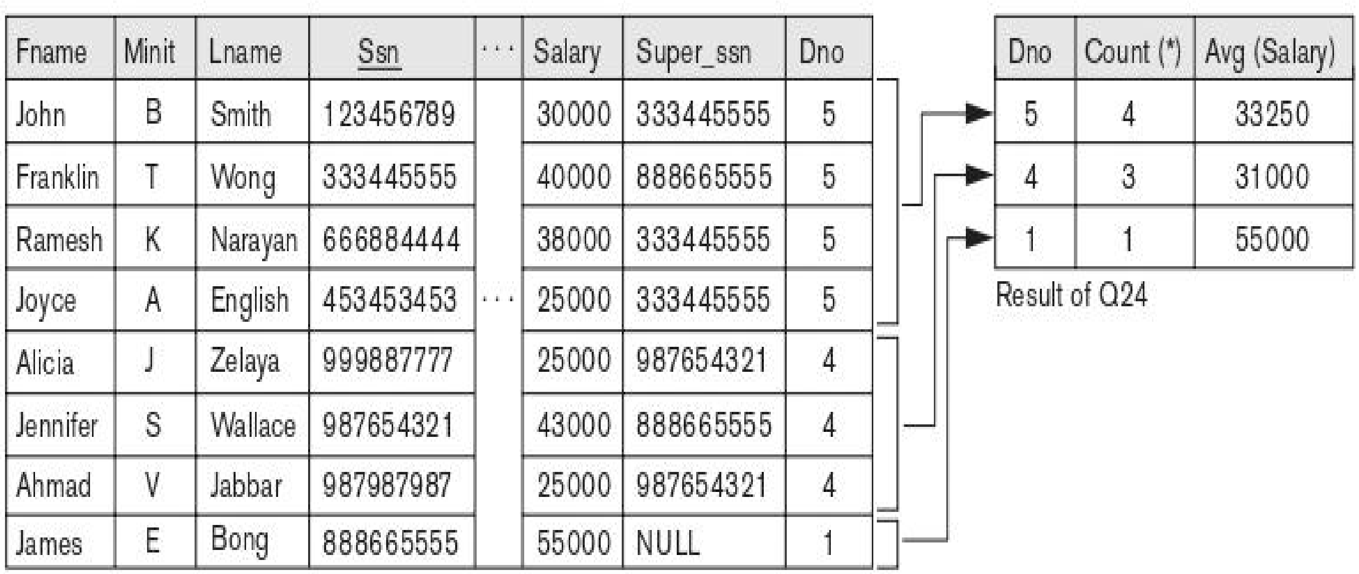
Q2: **SELECT COUNT** (*)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER **AND** DNAME = ‘Research’;

Grouping: The GROUP BY and HAVING Clauses:

In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the grouping attribute(s). SQL has a GROUP BY clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause.

Query: For each department, retrieve the department number, the number of employees in the department, and their average salary.

Q: **SELECT** Dno, **COUNT** (*), **AVG** (Salary)
FROM EMPLOYEE
GROUP BY Dno;



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
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

Views (Virtual Tables) in SQL

Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables. These other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database.

Specification of Views in SQL

In SQL, the command to specify a view is CREATE VIEW. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure when applied to the database schema.

```
V1:   CREATE VIEW   WORKS_ON1
      AS SELECT     Fname, Lname, Pname, Hours
      FROM          EMPLOYEE, PROJECT, WORKS_ON
      WHERE         Ssn = Essn AND Pno = Pnumber;
```

```
V2:   CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT Dname, COUNT (*), SUM (Salary)
      FROM DEPARTMENT, EMPLOYEE
      WHERE Dnumber = Dno
      GROUP BY Dname;
```

Figure: Two views specified on the database schema

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes,=.

If we do not need a view anymore, we can use the DROP VIEW command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement:

```
V1A:        DROP VIEW   WORKS_ON1;
```

View Implementation, View Update:

DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query into a query on the underlying base tables.
For example, the query QV1 would be automatically modified to the following query by the DBMS.

```
SELECT Fname, Lname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn = Essn AND Pno = Pnumber AND Pname = ‘ProductX’;
```

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time.

The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow.

In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways.

Consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of ‘John Smith’ from ‘ProductX’ to ‘ProductY’. This view update is shown in UV1:

```
UV1:   UPDATE     WORKS_ON1
       SET        Pname = ‘ProductY’
       WHERE      Lname = ‘Smith’ AND Fname = ‘John’
                  AND Pname = ‘ProductX’;
```

Views as Authorization Mechanisms:

SQL query authorization statements (GRANT and REVOKE), we present database security and authorization mechanisms. The views can be used to hide certain attributes or tuples from unauthorized users. Suppose a certain user is only allowed to see employee information for employees who work for department 5; then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself. This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW   DEPT5EMP  AS
SELECT         *
FROM          EMPLOYEE
WHERE         Dno = 5;
```

In a similar manner, a view can restrict a user to only see certain columns; for example, only the first name, last name, and address of an employee may be visible as follows:

```
CREATE VIEW    BASIC_EMP_DATA    AS
SELECT         Fname, Lname, Address
FROM           EMPLOYEE;
```

Database Stored Procedures

Database stored procedures, which are program modules that are stored by the DBMS at the database server.

Database Stored Procedures and Functions:

In our presentation of database programming techniques so far, there was an implicit assumption that the database application program was running on a client machine, or more likely at the application server computer in the middle-tier of a three-tier client-server architecture.

For many applications, it is sometimes useful to create database program modules—procedures or functions—that are stored and executed by the DBMS at the database server. These are historically known as database **stored procedures**.

Stored procedures are useful in the following circumstances:

1. If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.
2. Executing a program at the server can reduce data transfer and communication cost between the client and server in certain situations.
3. These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users via the stored procedures.

In general, many commercial DBMSs allow stored procedures and functions to be written in a general-purpose programming language. Alternatively, a stored procedure can be made of simple SQL commands such as retrievals and updates. The general form of declaring stored procedures is as follows:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;
```

The parameters and local declarations are optional, and are specified only if needed. For declaring a function, a return type is necessary, so the declaration form is:

```
CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body>;
```

If the procedure (or function) is written in a general-purpose programming language, it is typical to specify the language as well as a file name where the program code is stored. For example, the following format can be used:

```
CREATE PROCEDURE <procedure name> (<parameters>)
LANGUAGE <programming language name>
EXTERNAL NAME <file path name>;
```

The procedures and functions are stored persistently by the DBMS, it should be possible to call them from the various SQL interfaces and programming techniques.The **CALL** statement in the SQL standard can be used to invoke a stored procedure—either from an interactive interface or from embedded SQL or SQLJ. The format of the statement is as follows:

CALL <procedure or function name> (<argument list>);

****

UNIT - III

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.

Informal 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.

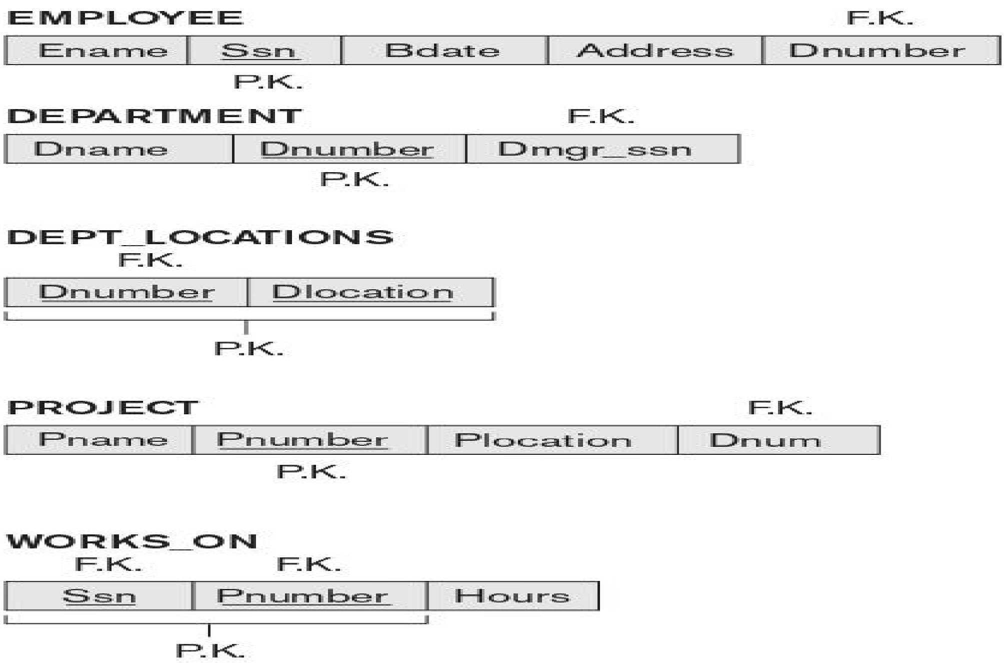
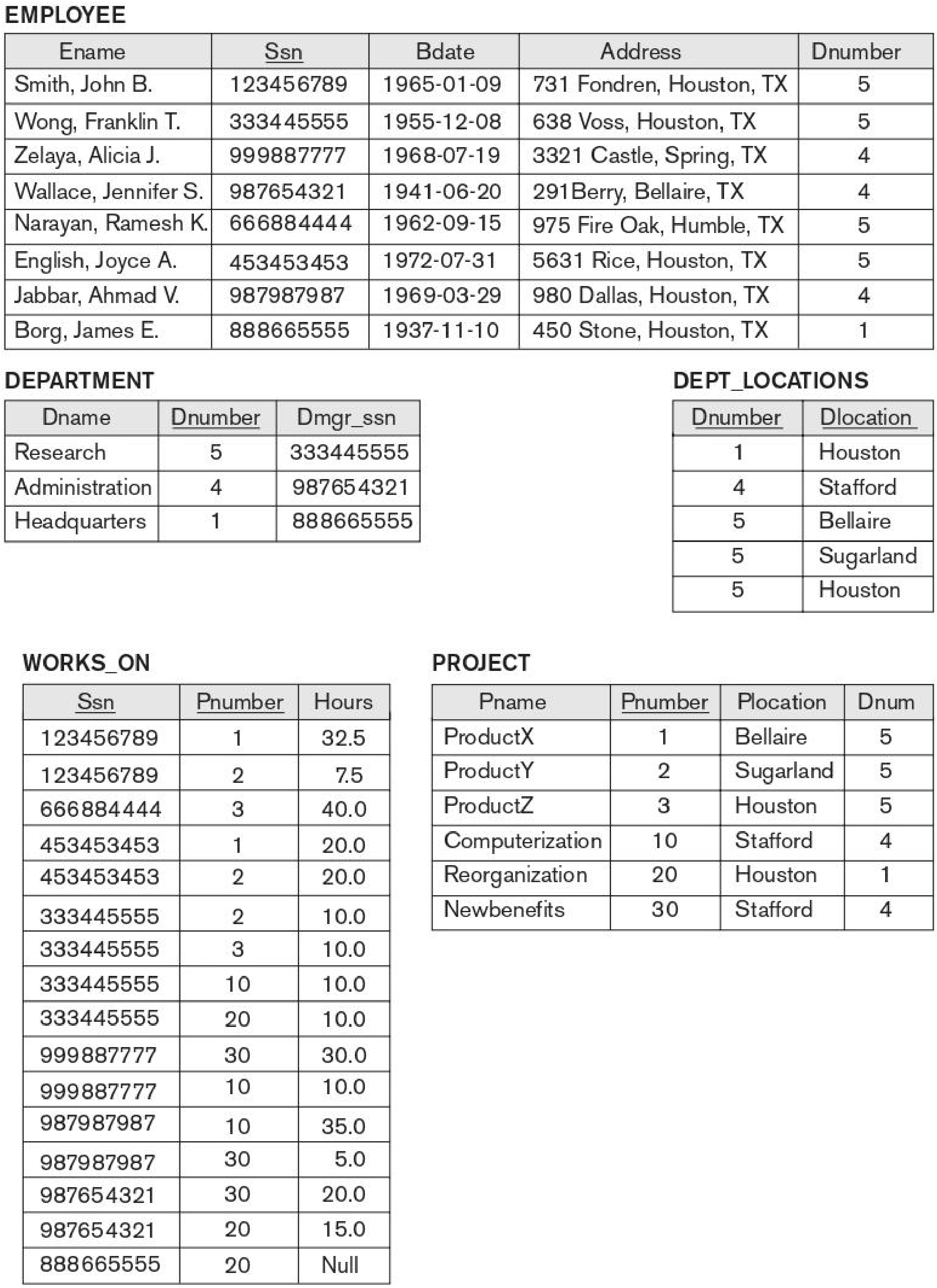


Figure: 2 Sample database state for the relational database schema in Figure 1



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

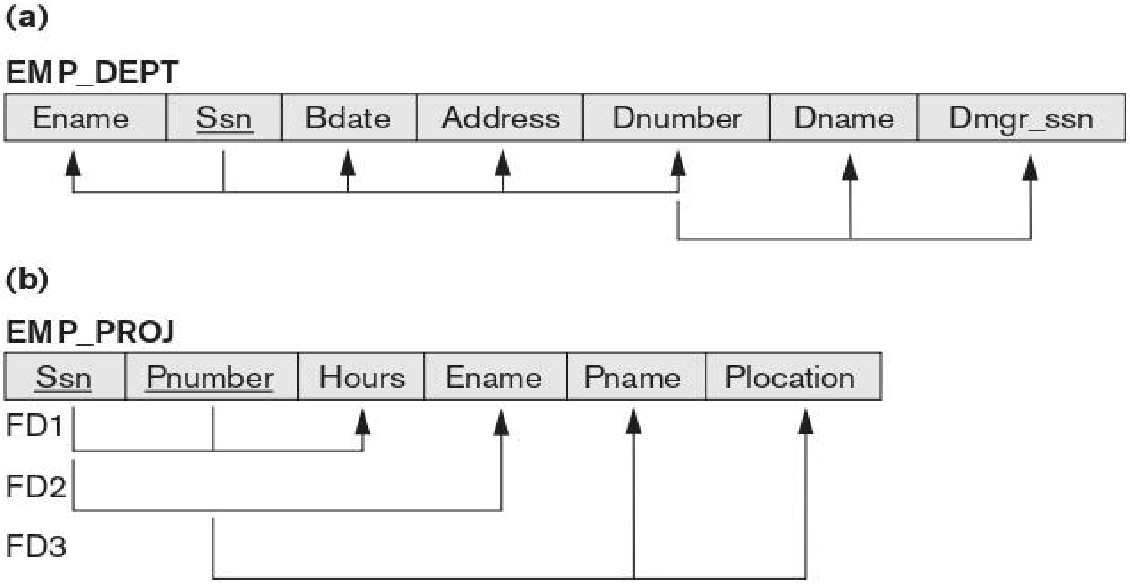
Ssn	Pnumber	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	Null

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).

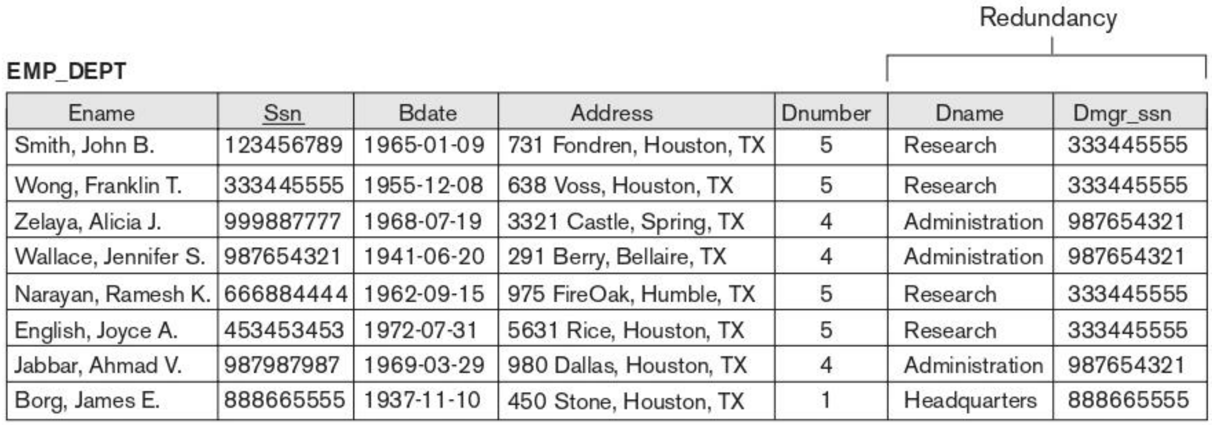


2. Redundant Information in Tuples and Update Anomalies:

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.

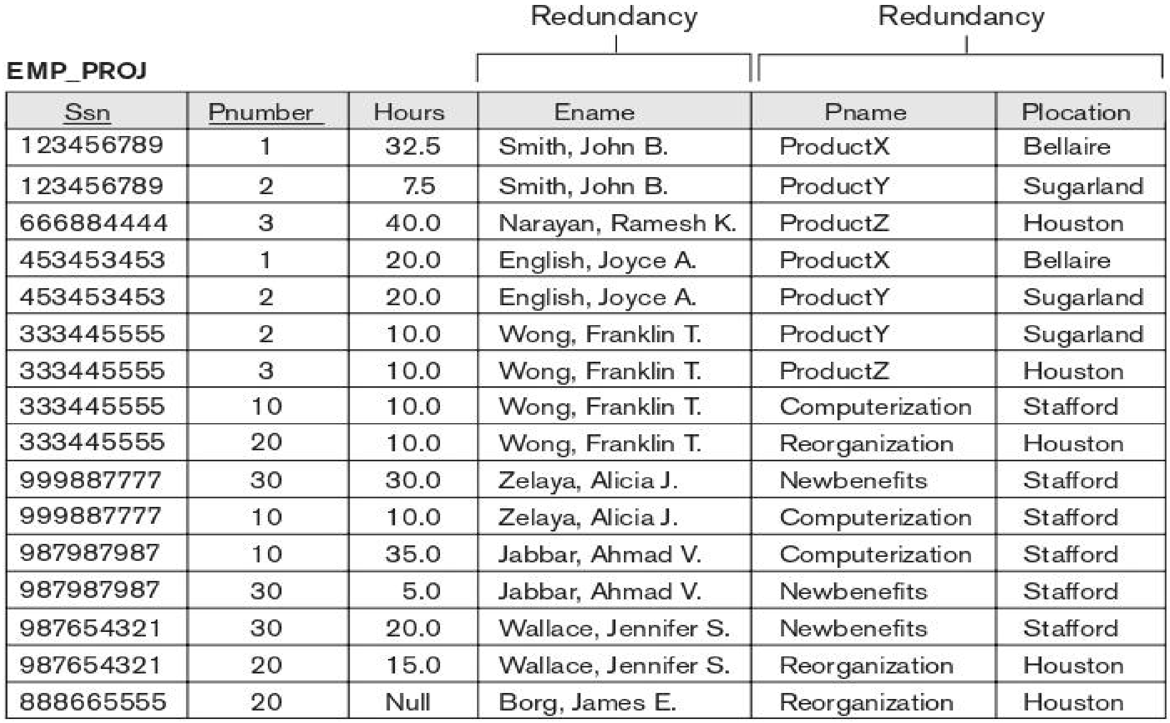
In Figure 3, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department. Only the department number (Dnumber) is 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.

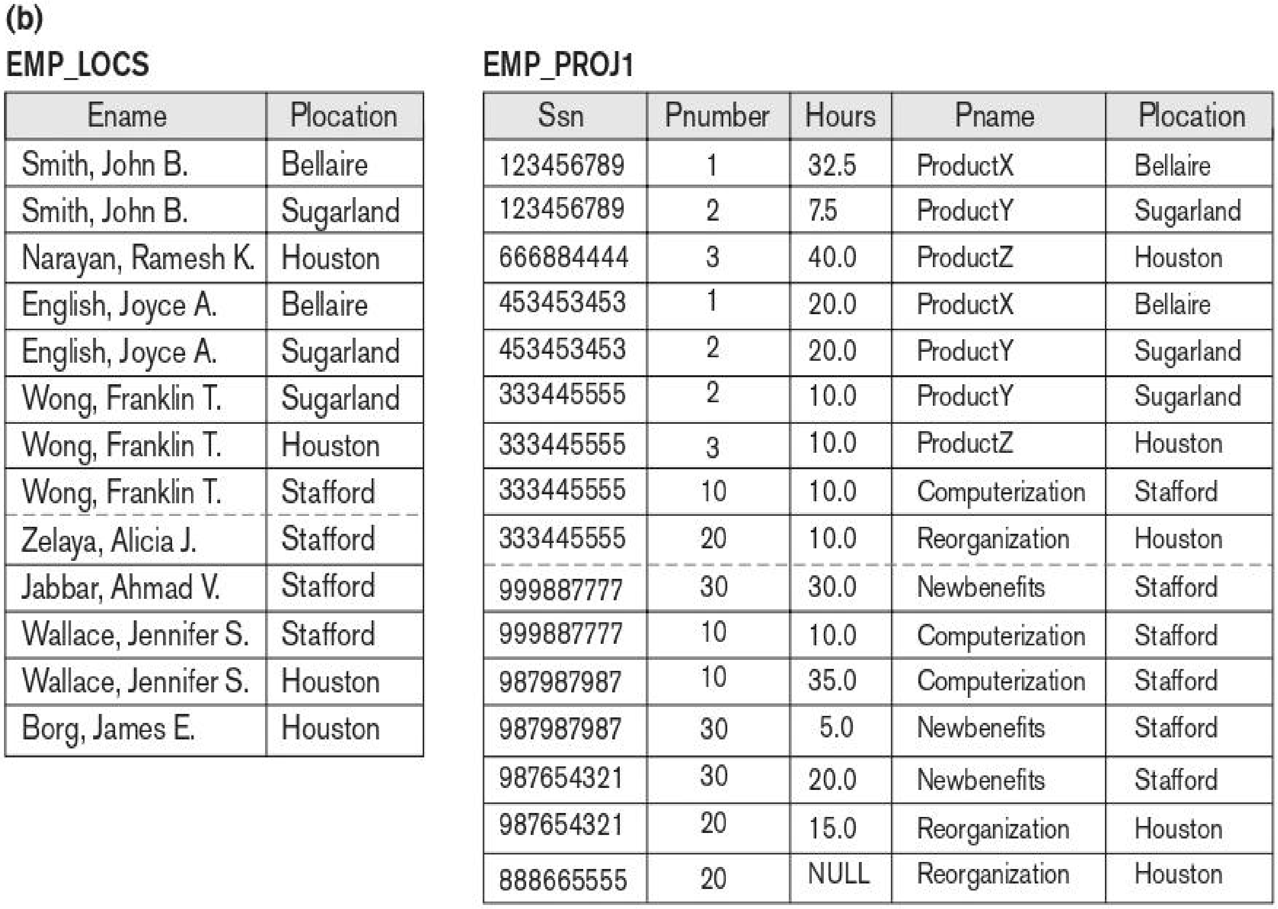
Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we state another guideline.

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)

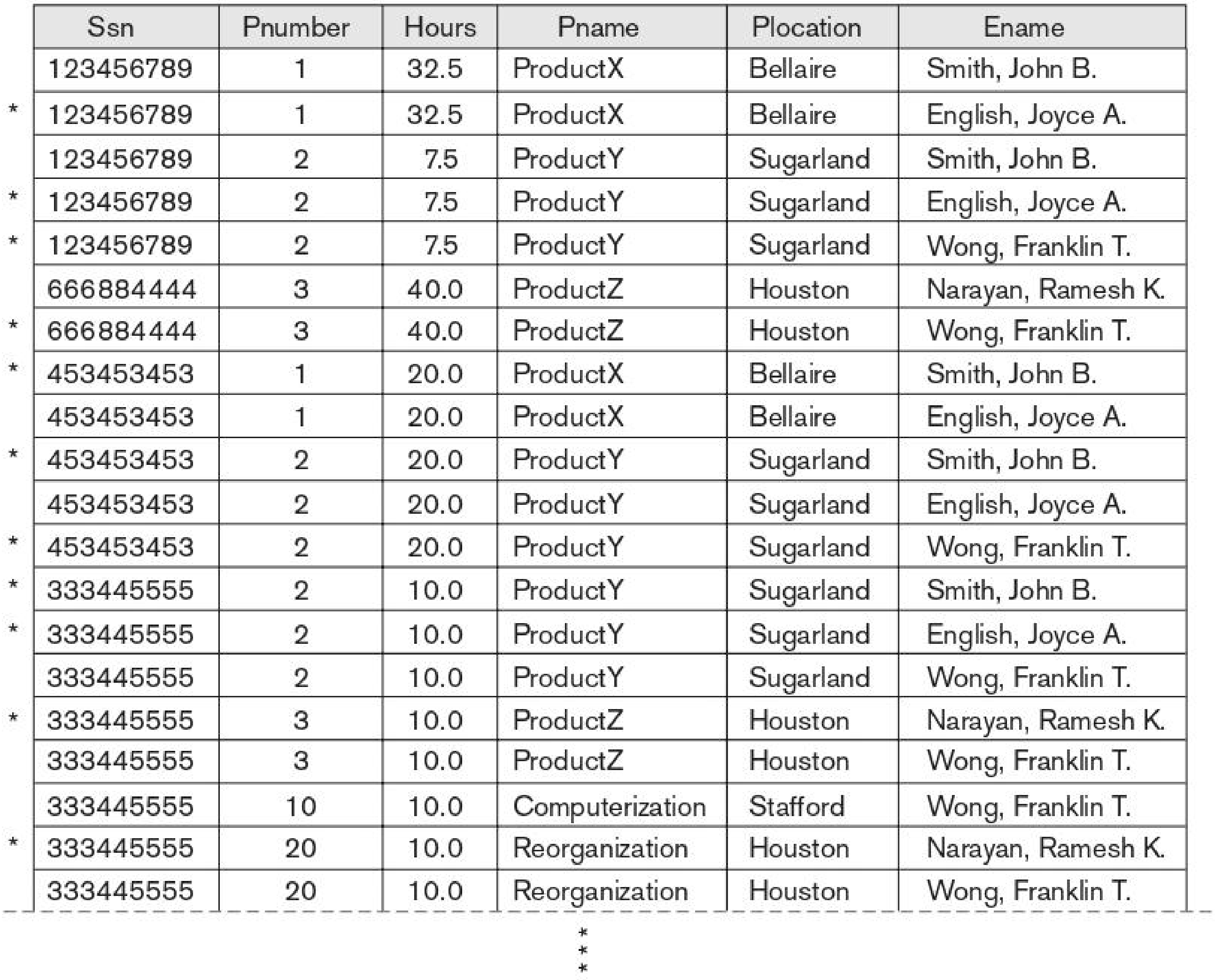
EMP_LOCS

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

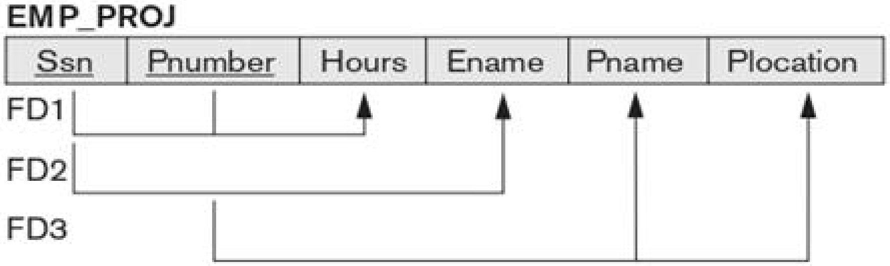
These functional dependencies specify that

(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.

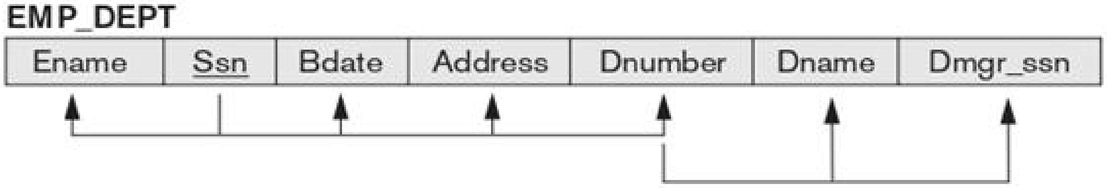
Figure: A relation state of TEACH with a possible functional dependency
TEXT → COURSE. However, TEACHER → COURSE,
TEXT → TEACHER and COURSE → TEXT are ruled out.

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.

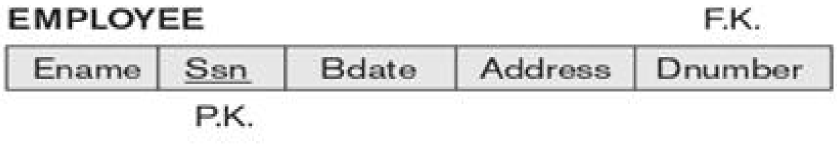
Definition: Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

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, … , An} is a set of attributes S ⊆ 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, … , Ak} of R, then K − {Ai} is not a key of R for any Ai, 1 ≤ i ≤ 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.

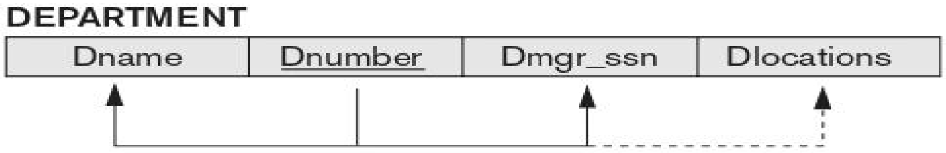
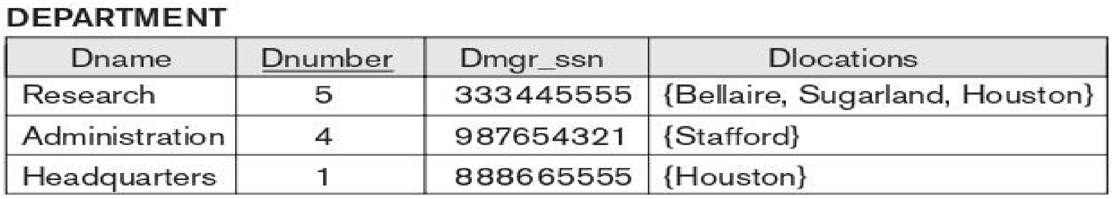


Figure:b) Sample state of relation DEPARTMENT.

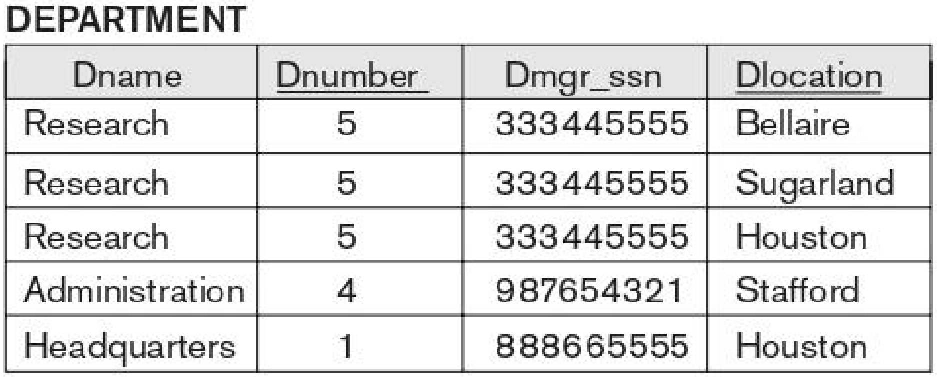


DEPARTMENT

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 .

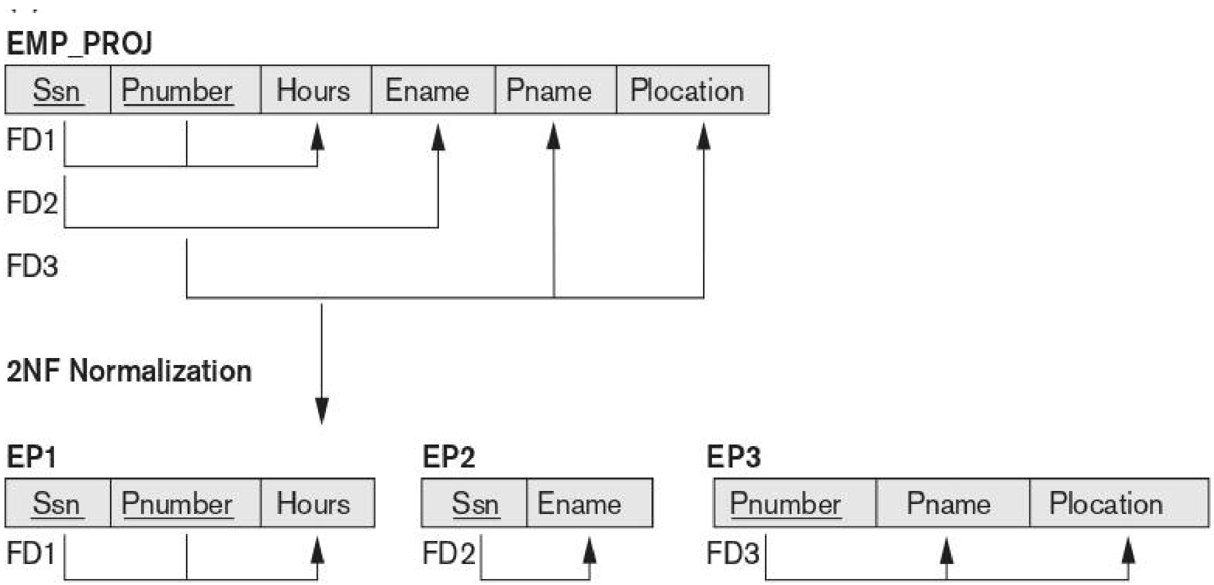
Definition: A relation schema R is in 2NF if every nonprime attribute A in R is ***fully functionally dependent*** on the primary key of R.

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:

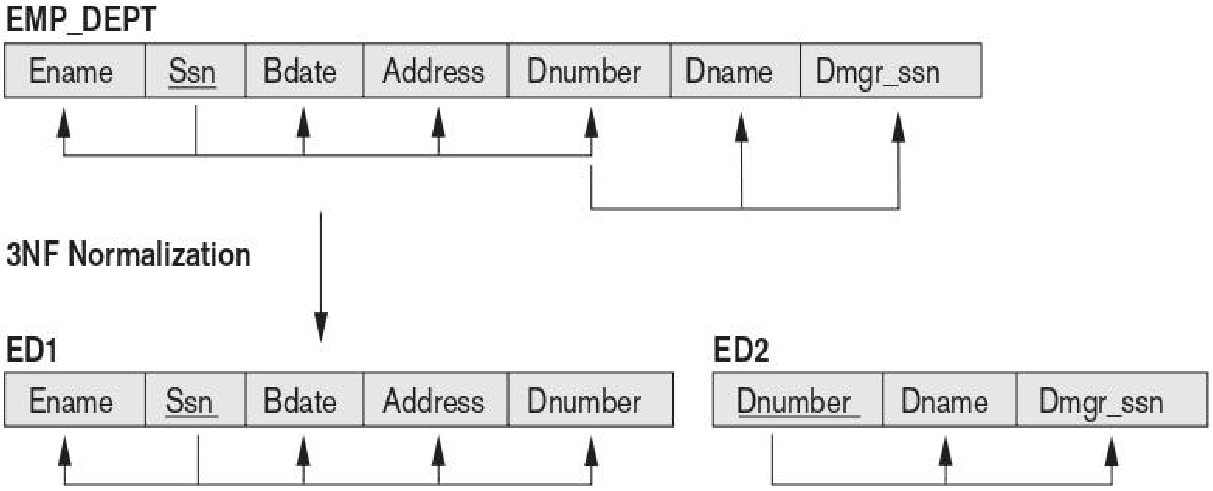
Third normal form (3NF) is based on the concept of ***transitive dependency***. A functional dependency X → Y in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold.

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.



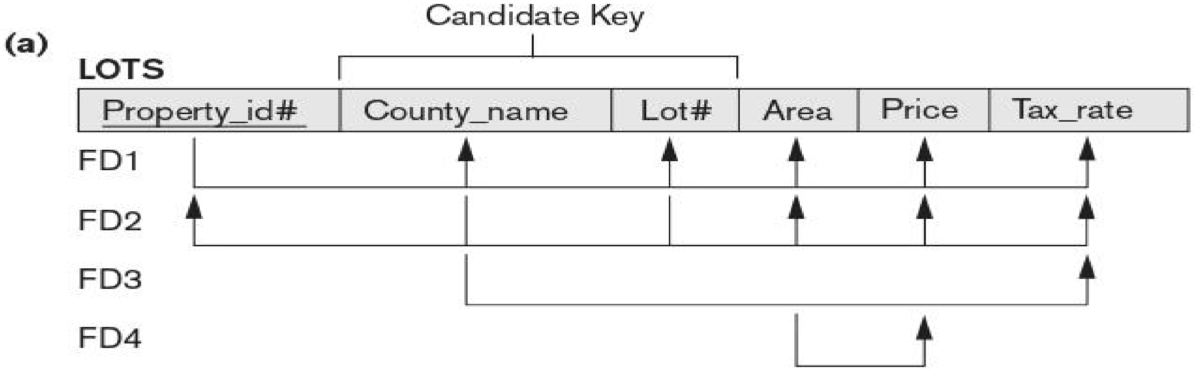
General Definitionsn of Second and Third Normal Forms

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#}.

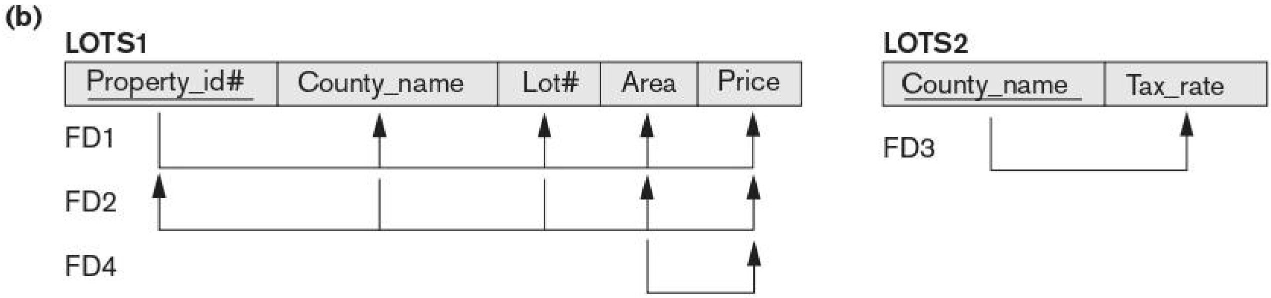
Figure (a): The LOTS relation with its functional dependencies FD1 through FD4



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.

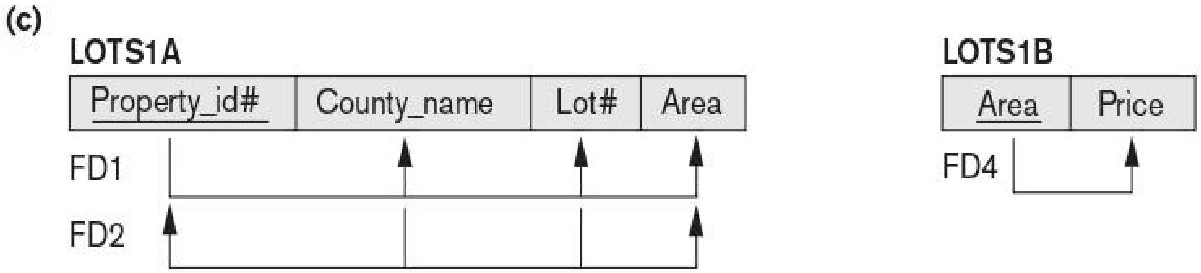
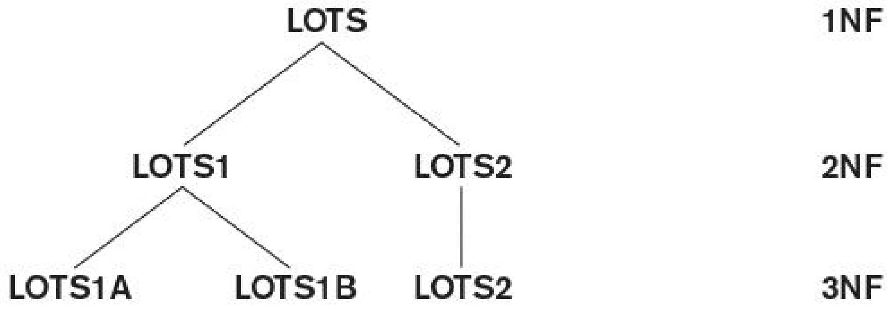


Figure: Progressive normalization of LOTS into a 3NF design.



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.

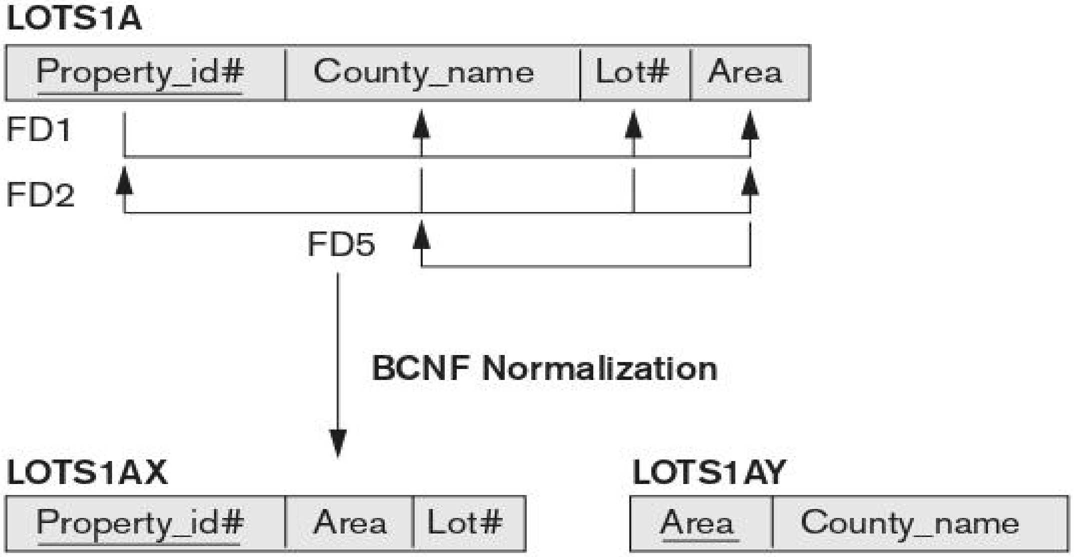
BOYCE-CODD Normal Form (BCNF)

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.

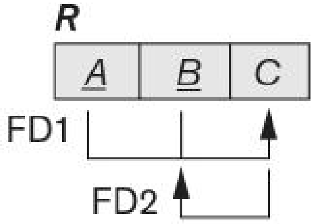
Figure: BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition.



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.



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 $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_k$ 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 $R = \{A_1, A_2, \dots , A_n\}$ into a decomposition $D = \{R_1, R_2, \dots , R_m\}$ 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 $D = \{R_1, R_2, \dots , R_m\}$ 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 $X \rightarrow Y$ in Q that violates BCNF;
       replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
     } ;
```

Each time through the loop in Algorithm 2, we decompose one relation schema Q that is not in BCNF into two relation schemas. 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.

Multivalued Dependency and Fourth Normal Form

Some relations have constraints that cannot be specified as functional dependencies and hence are not in violation of BCNF. To address this situation, the concept of **multivalued dependency** (MVD) was proposed and, based on this dependency, the **fourth normal form** was defined.

Multivalued dependencies are a consequence of first normal form (1NF) , which disallows an attribute in a tuple to have a set of values. If more than one multivalued attribute is present, the second option of normalizing the relation (see Section 14.3.4) introduces a multivalued dependency. Informally, whenever two independent 1:N relationships A:B and A:C are mixed in the same relation, R(A, B, C), an MVD may arise.

Formal Definition of Multivalued Dependency

Definition: A multivalued dependency X → Y specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t1 and t2 exist in r such that t1[X] = t2[X], then two tuples t3 and t4 should also exist in **r** with the following properties, where we use Z to denote (R − (X ∪ Y)):

- t3[X] = t4[X] = t1[X] = t2[X]
- t3[Y] = t1[Y] and t4[Y] = t2[Y]
- t3[Z] = t2[Z] and t4[Z] = t1[Z]

Whenever X →→ Y holds, we say that X multidetermines Y. Because of the symmetry in the definition, whenever X →→ Y holds in R, so does X →→ Z. Hence, X →→ Y implies X →→ Z and therefore it is sometimes written as X →→ Y|Z.

An MVD X →→ Y in R is called a trivial MVD if (a) Y is a subset of X, or (b) X ∪ Y = R. For example, the relation EMP_PROJECTS in (b) has the trivial MVD Ename →→ Pname and the relation EMP_DEPENDENTS has the trivial MVD Ename →→ Dname. An MVD that satisfies neither (a) nor (b) is called a nontrivial MVD.

If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure (a), the values 'X' and 'Y' of Pname are repeated with each value of Dname. This redundancy is clearly undesirable. However, the EMP schema is in BCNF because no functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP.

Fourth Normal Form:

Now present the definition of fourth normal form (**4NF**), which is violated when a relation has undesirable multivalued dependencies and hence can be used to identify and decompose such relations.

Definition: A relation schema R is in **4NF** with respect to a set of dependencies F if, for every nontrivial multivalued dependency X →→ Y in F+, X is a superkey for R.

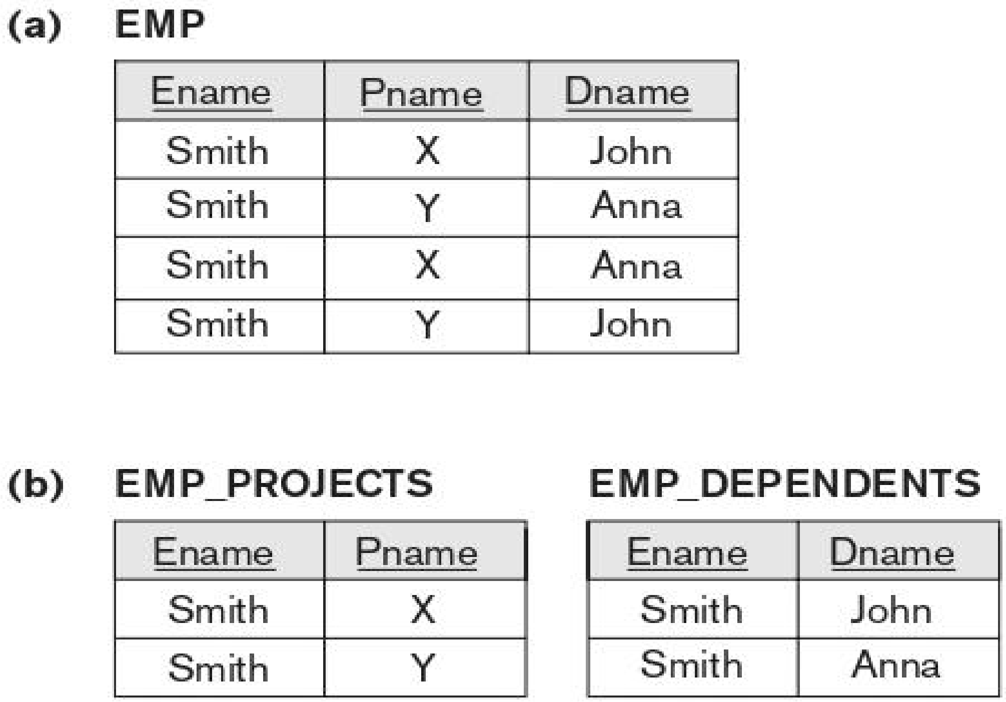
We can state the following points:

■ An all-key relation is always in BCNF since it has no FDs.
■ An all-key relation such as the EMP relation in Figure (a), which has no FDs but has the MVD Ename →→ Pname | Dname, is not in 4NF.
■ A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
■ The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure (a). EMP is not in 4NF because in the nontrivial MVDs Ename →→ Pname and Ename →→ Dname, and Ename is not a superkey of EMP.

We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure (b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename →→ Pname in EMP_PROJECTS and Ename →→ Dname in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

Figure: (a) The EMP relation with two MVDs: Ename →→ Pname and Ename →→ Dname. (b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.



(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

File Organization and Indexes

Introduction

Databases are stored physically as files of records, which are typically stored on magnetic disks. The organization of databases in storage and the techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called **indexes**. These structures are often referred to as **physical database file structures** and are at the physical level of the threeschema architecture.

The collection of data that makes up a computerized database must be stored physically on some computer **storage medium**. The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a *storage hierarchy* that includes two main categories:

■ **Primary storage:** This category includes storage media that can be operated on directly by the computer's central processing unit (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity.

■ **Secondary storage:** The primary choice of storage medium for online storage of enterprise databases has been magnetic disks. When used as a substitute for a disk drive, such memory is called a solid-state drive (SSD).

■ **Tertiary storage:** Optical disks (CD-ROMs, DVDs, and other similar storage media) and tapes are removable media used in today's systems as offline storage for archiving databases and hence come under the category called **tertiary storage**. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices.

1. Memory Hierarchies and Storage Devices:

In a modern computer system, data resides and is transported throughout a hierarchy of storage media. At the ***primary storage level***, the memory hierarchy includes, at the most expensive end, **cache memory**, which is a static RAM (random access memory). Cache memory is typically used by the CPU to speed up execution of program instructions. The next level of primary storage is DRAM (dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called **main memory**. The advantage of DRAM is its low cost.

At the ***secondary and tertiary storage level***, the hierarchy includes magnetic disks; **mass storage** in the form of CD-ROM (compact disk–read-only memory) and DVD (digital video disk or digital versatile disk) devices; and finally tapes at the least expensive end of the hierarchy.

The storage capacity is measured in kilobytes (Kbyte or 1,000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1,000 GB). The word ***petabyte*** (1,000 terabytes or 10**15 bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Flash Memory: Between DRAM and magnetic disk storage, another form of memory, flash memory, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (electrically erasable programmable read-only memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memories come in two types called NAND and NOR flash based on the type of logic circuits used.

Optical Drives: The most popular form of optical removable storage is CDs (compact disks) and DVDs. CDs have a 700-MB capacity whereas DVDs have capacities ranging from 4.5 to 15 GB. CD-ROM(compact disk – read only memory) disks store data optically and are read by a laser.

Magnetic Tapes: Finally, **magnetic tapes** are used for archiving and backup storage of data. **Tape jukeboxes**—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as **tertiary storage** to hold terabytes of data.

2. Storage Organization of Databases

Databases typically store large amounts of data that must persist over long periods of time, and hence the data is often referred to as **persistent data**. Parts of this data are accessed and processed repeatedly during the storage period. This contrasts with the notion of **transient data**, which persists for only a limited time during program execution.

Most databases are stored permanently (or persistently) on magnetic disk secondary storage, for the following reasons:

■ Generally, databases are too large to fit entirely in main memory.

■ The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as **nonvolatile storage**, whereas main memory is often called **volatile storage**.

■ The cost of storage per unit of data is an order of magnitude less for disk secondary storage than for primary storage.

Secondary Storage Devices

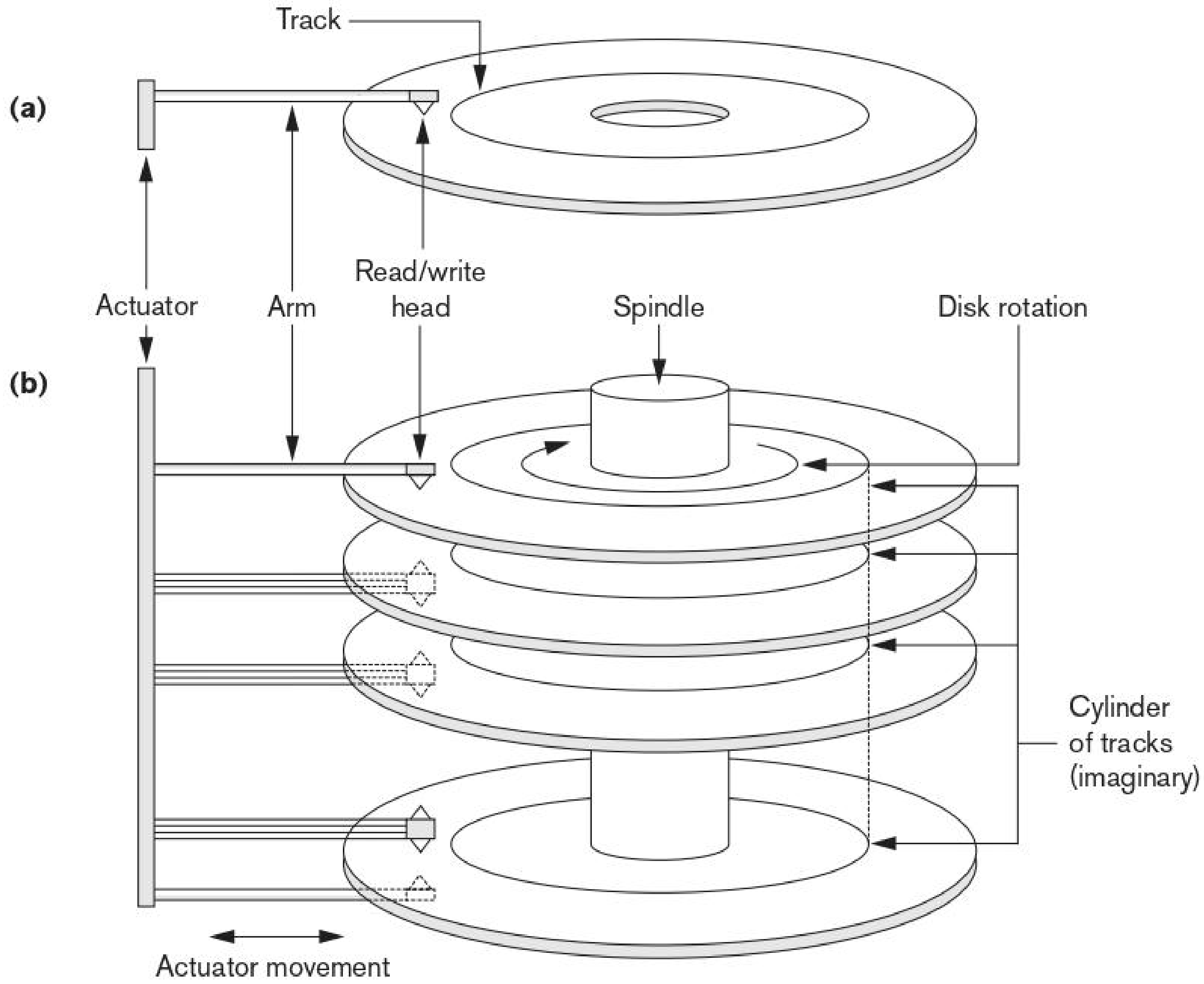
1. Hardware Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The device that holds the disks is referred to as a **hard disk drive**, or **HDD**. The most basic unit of data on the disk is a single **bit** of information. The area represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into **bytes** (or **characters)**. Byte sizes are typically 4 to 8 bits, depending on the computer and the device; 8 bits is the most common. The **capacity** of a disk is the number of bytes it can store, which is usually very large.

All disks are made of magnetic material shaped as a thin circular disk, as shown in Figure (a), and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on one of its surfaces only and **double-sided** if both surfaces are used.

To increase storage capacity, disks are assembled into a disk pack, as shown in Figure (b), which may include many disks and therefore many surfaces. The two most common form factors are 3.5 and 2.5 inch diameter. Information is stored on a disk surface in concentric circles of small width,5 each having a distinct diameter. Each circle is called a **track**. In disk packs, tracks with the same diameter on the various surfaces are called a **cylinde**r because of the shape they would form if connected in space.

Figure: (a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.



A track usually contains a large amount of information, it is divided into smaller **blocks** or **sectors**. The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed. One type of sector organization, as shown in the following Figure (1), calls a portion of a track that subtends a fixed angle at the center a sector. Several other sector organizations are possible, one of which is to have the sectors subtend smaller angles at the center as one moves away, thus maintaining a uniform density of recording, as shown in the following Figure (2).

Figure 1: Sectors subtending a fixed angle.

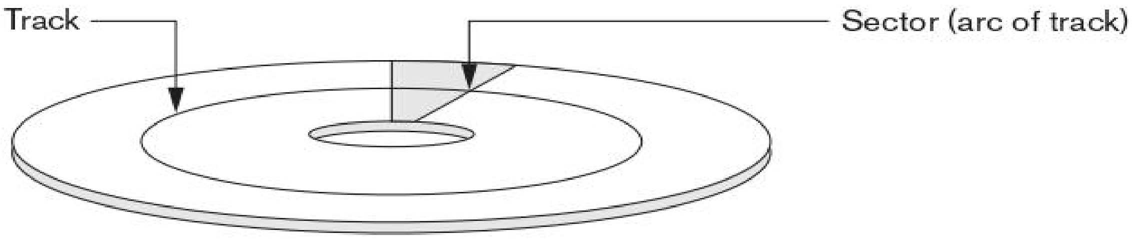
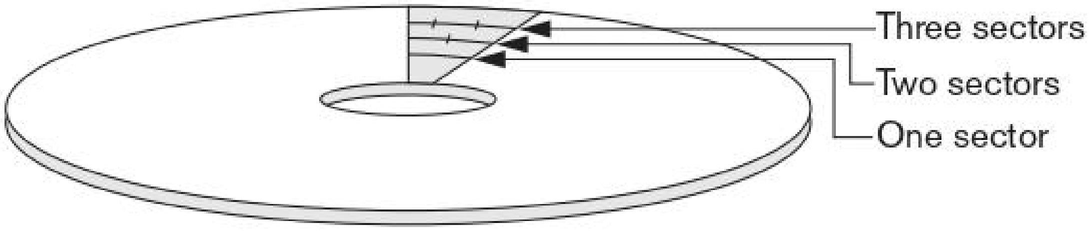


Figure 2: Sectors maintaining a uniform recording density.



2. Making Data Access More Efficient on Disk

Some of the commonly used techniques to make accessing data more efficient on HDDs:

1. **Buffering of data:** The electromechanical device such as an HDD, which is inherently slower, buffering of data is done in memory so that new data can be held in a buffer while old data is processed by an application.

2. **Proper organization of data on disk:** Given the structure and organization of data on disk, it is advantageous to keep related data on contiguous blocks; when multiple cylinders are needed by a relation, contiguous cylinders should be used.

3. **Reading data ahead of request:** To minimize seek times, whenever a block is read into the buffer, blocks from the rest of the track can also be read even though they may not have been requested yet.

3. SolidState Device (SSD) Storage

This type of storage is sometimes known as flash storage because it is based on the flash memory technology. The recent trend is to use flash memories as an intermediate layer between main memory and secondary rotating storage in the form of magnetic disks (HDDs). Since they resemble disks in terms of the ability to store data in secondary storage without the need for continuous power supply, they are called **solid-state disks** or **solid-state drives (SSDs**). SSDs in general terms first and then comment on their use at the enterprise level, where they are sometimes referred to as **enterprise flash drives** (EFDs).

The main component of an SSD is a controller and a set of interconnected flash memory cards. Use of NAND flash memory is most common. In addition to flash memory, DRAM-based SSDs are also available. They are costlier than flash memory. As an example of an enterprise level SSD, we can consider CISCO's UCS (Unified Computing System) Invicta series SSDs.

4. Magnetic Tape Storage Devices

Disks are random access secondary storage devices because an arbitrary disk block may be accessed at random once we specify its address. Magnetic tapes are sequential access devices; to access the **n**th block on tape, first we must scan the preceding n – 1 blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audiotapes or videotapes. A tape drive is required to read the data from or write the data to a tape reel. Usually, each group of bits that forms a byte is stored across the tape, and the bytes themselves are stored consecutively on the tape. A read/write head is used to read or write data on tape. Data records on tape are also stored in blocks.

The main characteristic of a tape is its requirement that we access the data blocks in **sequential order**. To get to a block in the middle of a reel of tape, the tape is mounted and then scanned until the required block gets under the read/write head.

Tapes serve a very important function— backing up the database. One reason for backup is to keep copies of disk files in case the data is lost due to a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape.

Buffering of Blocks

When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

Figure 1 illustrates how two processes can proceed in parallel. Processes A and B are running **concurrently** in an i**nterleaved** fashion, whereas processes C and D are running **concurrently** in a **parallel** fashion. When a single CPU controls multiple processes, parallel execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when processes can run concurrently in a parallel fashion.

Figure 1: Interleaved concurrency versus parallel execution.

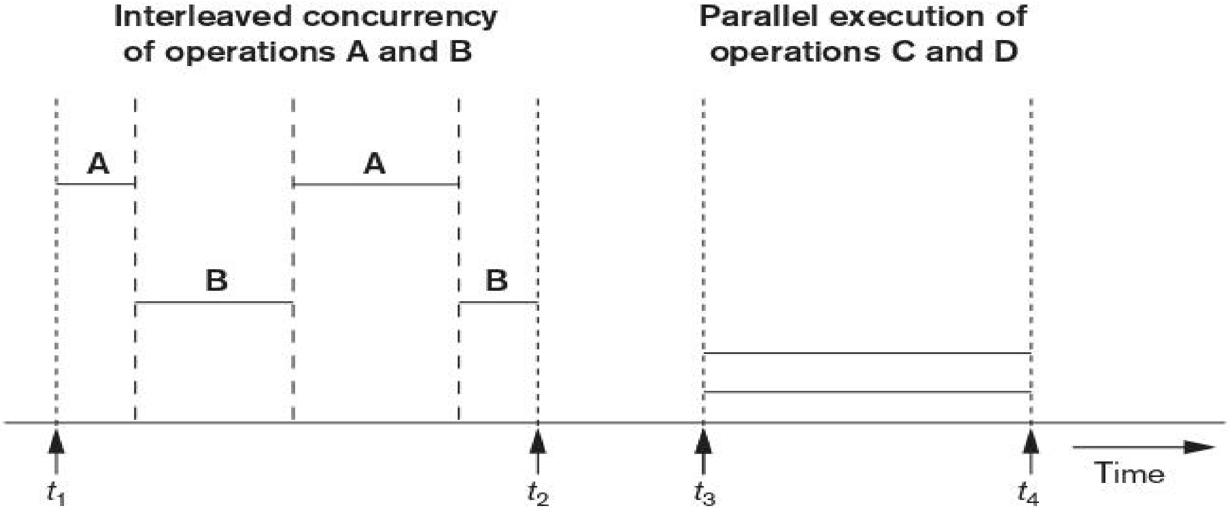
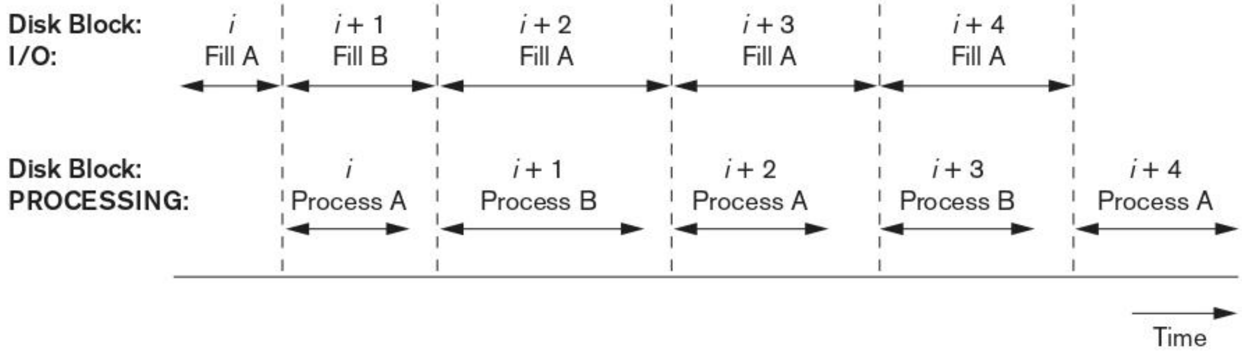

Interleaved concurrency of operations A and B
Parallel execution of operations C and D
A
A
B
B
t_1
t_2
t_3
t_4
Time

Figure 2 illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer. The CPU can start processing a block once its transfer to main memory is completed; at the same time, the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called **double buffering** and can also be used to read a continuous stream of blocks from disk to memory. Double buffering permits continuous reading or writing of data on consecutive disk blocks.

Figure 2: Use of two buffers, A and B, for reading from disk.



1. **Buffer Management**

Buffer management and Replacement Strategies: For most large database files containing millions of pages, it is not possible to bring all of the data into main memory at the same time. The actual management of buffers and decisions about what buffers to use to place a newly read page in the buffer is a more complex process. We use the term **buffer** to refer to a part of main memory that is available to receive blocks or pages of data from disk.

Buffer manager is a software component of a DBMS that responds to requests for data and decides what buffer to use and what pages to replace in the buffer to accommodate the newly requested blocks. The buffer manager views the available main memory storage as a **buffer pool**, which has a collection of pages.

There are two kinds of buffer managers; the first kind controls the main memory directly, as in most RDBMSs. The second kind allocates buffers in virtual memory, which allows the control to transfer to the operating system (OS).

The overall goal of the buffer manager is twofold: (1) to maximize the probability that the requested page is found in main memory, and (2) in case of reading a new disk block from disk, to find a page to replace that will cause the least harm in the sense that it will not be required shortly again.

2. Buffer Replacement Strategies:

The following are some popular replacement strategies that are similar to those used elsewhere, such as in operating systems:

1. **Least recently used (LRU):** The strategy here is to throw out that page that has not been used (read or written) for the longest time. This requires the buffer manager to maintain a table where it records the time every time a page in a buffer is accessed. Whereas this constitutes an overhead, the strategy works well because for a buffer that is not used for a long time, its chance of being accessed again is small.

2. **Clock policy:** This is a round-robin variant of the LRU policy. Imagine the buffers are arranged like a circle similar to a clock. Each buffer has a flag with a 0 or 1 value. Buffers with a 0 are vulnerable and may be used for replacement and their contents read back to disk. Buffers with a 1 are not vulnerable. When a block is read into a buffer, the flag is set to 1. When the buffer is accessed, the flag is set to 1 also. The clock hand is positioned on a "current buffer." When the buffer manager needs a buffer for a new block, it rotates the hand until it finds a buffer with a 0 and uses that to read and place the new block.

3. **First-in-first-out (FIFO):** Under this policy, when a buffer is required, the one that has been occupied the longest by a page is used for replacement. Under this policy, the manager notes the time each page gets loaded into a buffer. Although FIFO needs less maintenance than LRU, it can work counter to desirable behavior. A block that remains in the buffer for a long time because it is needed continuously.

LRU and clock policies are not the best policies for database applications if they require sequential scans of data and the file cannot fit into the buffer at one time.

Placing File Records on Disk

- **Records and Record Types**

Data is usually stored in the form of **records**. Each record consists of a collection of related data **values** or **items**, where each value is formed of one or more bytes and corresponds to a particular **field** of the record. Records usually describe entities and their attributes. For example, an **EMPLOYEE** record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as ***Name, Birth_date, Salary, or Supervisor***. A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition. A **data type**, associated with each field, specifies the types of values a field can take.

The data type of a field is usually one of the standard data types used in programming. These include numeric (integer, long integer, or floating point), string of characters (fixed-length or varying), Boolean (having 0 and 1 or TRUE and FALSE values only), and sometimes specially coded date and time data types. An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 10 bytes (assuming a format of YYYY-MM-DD), and a fixed-length string of k characters k bytes. For example, an EMPLOYEE record type may be defined—using the C programming language notation—as the following structure:

```
struct employee
{
    char name[30];
    char ssn[9];
    int salary;
    int job_code;
    char department[20];
} ;
```

- ## Files, Fixed-Length Records, and Variable-Length Records

A **file** is a ***sequence*** of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records**. If different records in the file have different sizes, the file is said to be made up of **variable-length records**. A file may have variable-length records for several reasons:

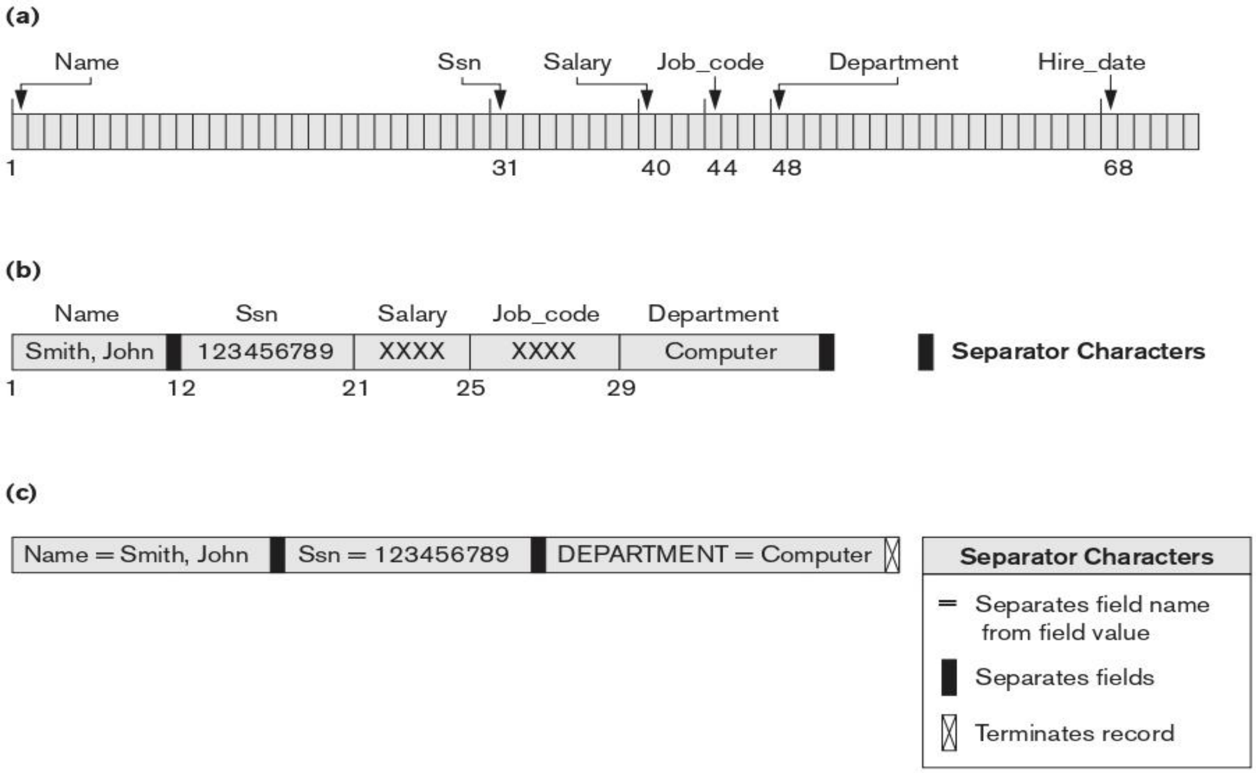
■ The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the Name field of EMPLOYEE can be a variable-length field.

■ The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a **repeating group**.

■ The file records are of the same record type, but one or more of the fields are **optional**; that is, they may have values for some but not all of the file records (**optional fields).**

■ The file contains records of different record types and hence of varying size (**mixed file**). This would occur if related records of different types were clustered (placed together) on disk blocks; for example, the GRADE_REPORT records of a particular student may be placed following that STUDENT's record.

Figure: Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

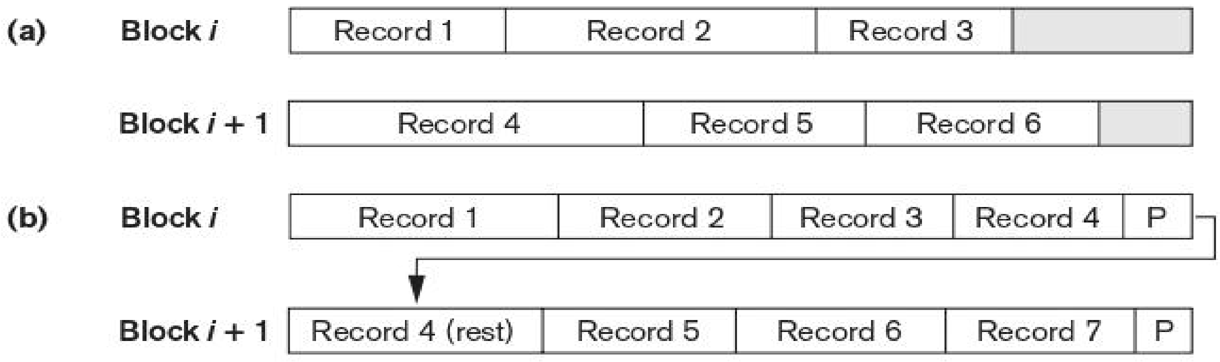


• Record Blocking and Spanned versus Unspanned Records

The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.

To utilize the unused space, we can store part of a record on one block and the rest on another. A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called **spanned** because records can span more than one block. Whenever a record is larger than a block, we must use a spanned organization. If records are not allowed to cross block boundaries, the organization is called **unspanned.**

Figure: Types of record organization. (a) Unspanned. (b) Spanned.



• Allocating File Blocks on Disk

There are several standard techniques for allocating the blocks of a file on disk. In **contiguous allocation**, the file blocks are allocated to consecutive disk blocks. This makes reading the whole file very fast using double buffering, but it makes expanding the file difficult. In **linked allocation**, each file block contains a pointer to the next file block. This makes it easy to expand the file but makes it slow to read the whole file. A combination of the two allocates **clusters** of consecutive disk blocks, and the clusters are linked. Clusters are sometimes called **file segments** or **extents**. Another possibility is to use indexed allocation, where one or more **index blocks** contain pointers to the actual file blocks. It is also common to use combinations of these techniques.

- File Headers

A **file header** or **file descriptor** contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

Operations on Files

Operations on files are usually grouped into **retrieval operations** and **update operations.** The former do not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to **select** one or more records for retrieval, deletion, or modification based on a **selection condition (**or **filtering condition**), which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields Name, Ssn, Salary, Job_code, and Department. A **simple selection condition** may involve an equality comparison on some field value—for example, (Ssn = ‘123456789’) or (Department = ‘Research’). More complex conditions can involve other types of comparison operators, such as > or ≥ ; an example is (Salary ≥ 30000). The general case is to have an arbitrary Boolean expression on the fields of the file as the selection condition.

Search operations on files are generally based on simple selection conditions. A complex condition must be decomposed by the DBMS (or the programmer) to extract a simple condition that can be used to locate the records on disk. For example, we may extract the simple condition (Department = ‘Research’) from the complex condition ((Salary ≥ 30000) AND (Department = ‘Research’)); each record satisfying (Department = ‘Research’) is located and then tested to see if it also satisfies (Salary ≥ 30000).

Actual operations for locating and accessing file records vary from system to system. In the following list, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access records by using these commands, so we sometimes refer to program variables in the following descriptions:

- **Open**: Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.
- **Reset.** Sets the file pointer of an open file to the beginning of the file.
- **Find (or Locate):** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the current record. Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.
- **Read (or Get):** Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.
- **FindNext:** Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record.
- **Delete:** Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
- **Modify:** Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.
- **Insert.** Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.
- **Close:** Completes the file access by releasing the buffers and performing any other needed cleanup operations.

In database systems, additional set-at-a-time higher-level operations may be applied to a file. Examples of these are as follows:

- **FindAll.** Locates all the records in the file that satisfy a search condition.
- **Find (or Locate) n.** Searches for the first record that satisfies a search condition and then continues to locate the next n − 1 records satisfying the same condition. Transfers the blocks containing the n records to the main memory buffer (if not already there).
- **FindOrdered.** Retrieves all the records in the file in some specified order.

- **Reorganize.** Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

At this point, it is worthwhile to note the difference between the terms file organization and access method. A file organization refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked.

Usually, we expect to use some search conditions more than others. Some files may be static, meaning that update operations are rarely performed; other, more dynamic files may change frequently, so update operations are constantly applied to them.

Hashing Techniques

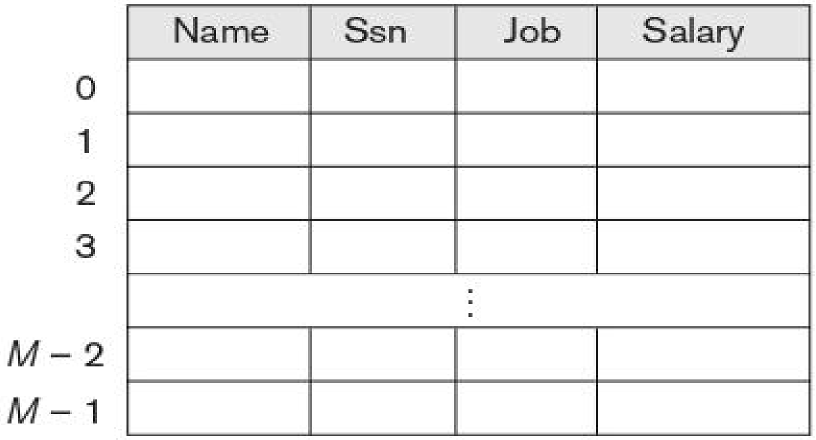
Another type of primary file organization is based on hashing, which provides very fast access to records under certain search conditions. This organization is usually called a **hash file**. The search condition must be an equality condition on a single field, called the **hash field**. In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**. The idea behind hashing is to provide a function h, called a **hash function** or **randomizing function.**

Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field.

1. Internal Hashing

For internal files, hashing is typically implemented as a hash table through the use of an array of records. Suppose that the array index range is from 0 to M – 1, as shown in Figure 16.8(a); then we have M slots whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and M − 1.

Figure: Array of M positions for use in internal hashing.



	Name	Ssn	Job	Salary
0				
1				
2				
3				
	⋮			
$M-2$				
$M-1$				

Other hashing functions can be used. One technique, called folding, involves applying an arithmetic function such as addition or a logical function such as exclusive or to different portions of the hash field value to calculate the hash address. Another technique involves picking some digits of the hash field value—for instance, the third, fifth, and eighth digits—to form the hash address.

A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called **collision resolution**. There are numerous methods for collision resolution, including the following:

■ **Open addressing:** Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.

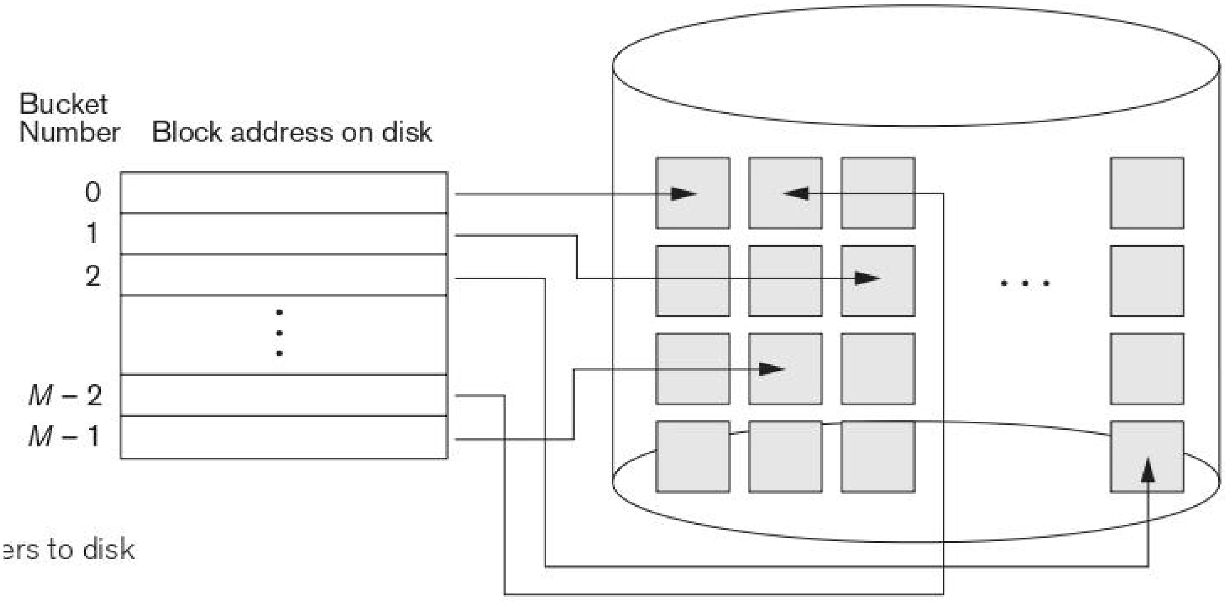
■ **Chaining:** For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. Additionally, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location.

■ **Multiple hashing**: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

2. External Hashing for Disk Files

Hashing for disk files is called **external hashing**. To suit the characteristics of disk storage, the target address space is made of **buckets,** each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function maps a key into a relative bucket number rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in Figure.

Figure: Matching bucket numbers to disk block addresses



The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket, as shown in Figure. The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block.

Figure: Handling overflow for buckets by chaining.

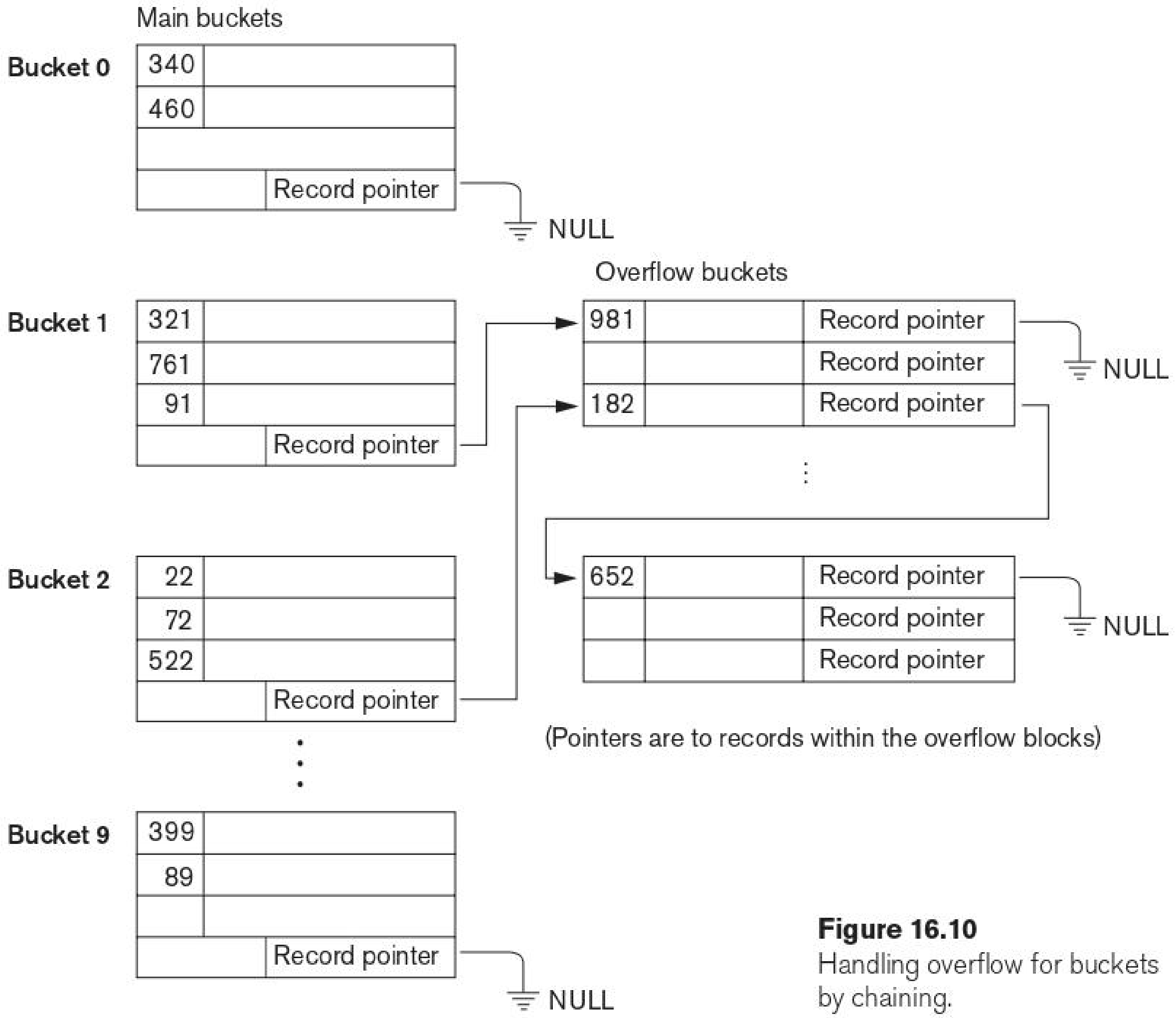


Figure 16.10
Handling overflow for buckets by chaining.

3. Hashing Techniques That Allow Dynamic File Expansion

A major drawback of the static hashing scheme is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. There are three schemes to remedy this situation. The first scheme—**extendible hashing**—stores an access structure in addition to the file. The second technique, called **linear hashing,** does not require additional access structures. Another scheme, called **dynamic hashing**, uses an access structure based on binary tree data structures.

Extendible Hashing: In extendible hashing, proposed by Fagin (1979), a type of directory—an array of 2^d bucket addresses—is maintained, where d is called the global depth of the directory. A local depth d′—stored with each bucket—specifies the number of bits on which the bucket contents are based. Figure shows a directory with global depth d = 3.

Figure: Structure of the extendible hashing scheme.

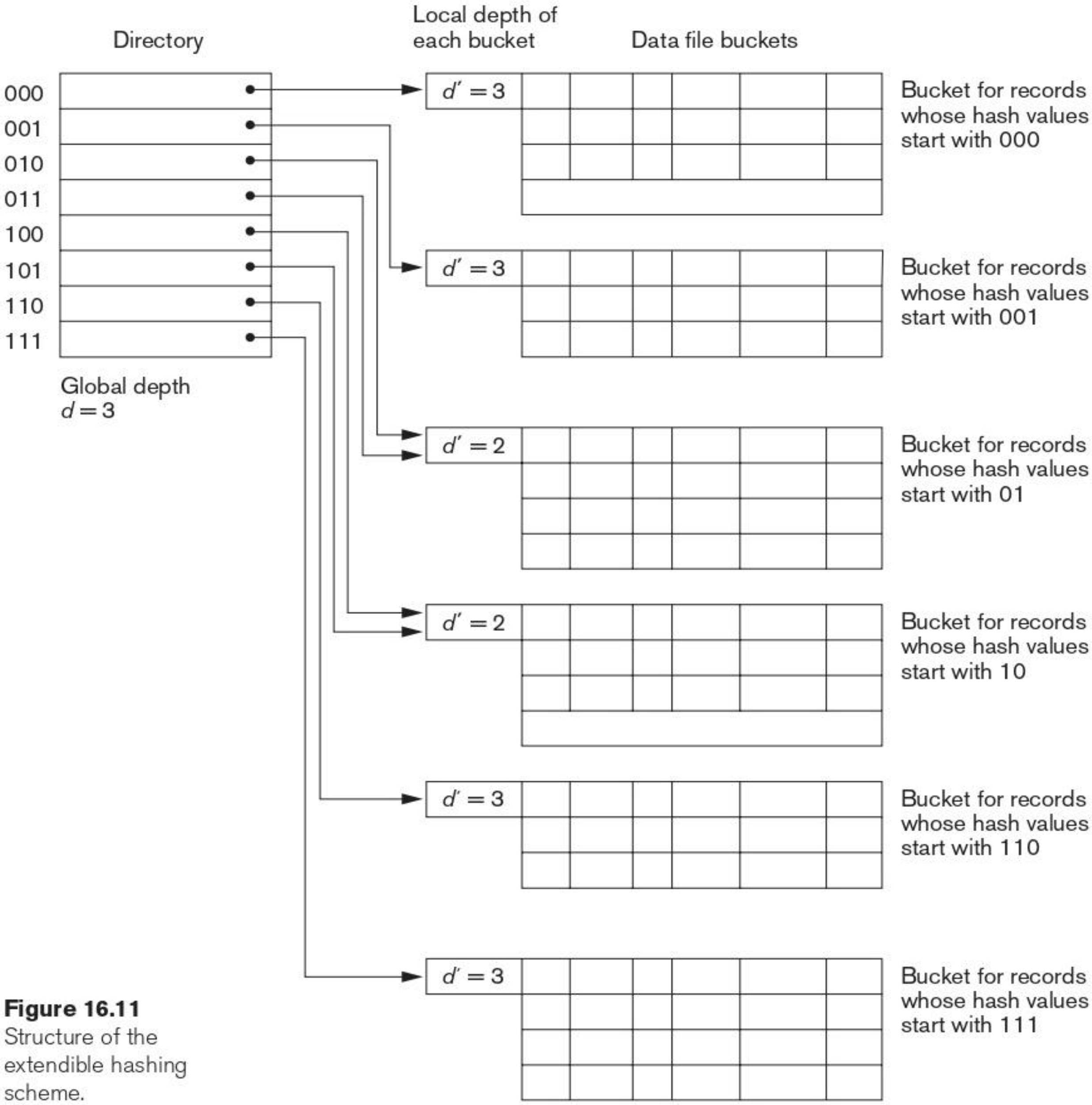

Directory
Local depth of each bucket
Data file buckets
000
001
010
011
100
101
110
111
Global depth d = 3
d′ = 3
Bucket for records whose hash values start with 000
d′ = 3
Bucket for records whose hash values start with 001
d′ = 2
Bucket for records whose hash values start with 01
d′ = 2
Bucket for records whose hash values start with 10
d′ = 3
Bucket for records whose hash values start with 110
d′ = 3
Bucket for records whose hash values start with 111

Figure 16.11
Structure of the extendible hashing scheme.

The main advantage of extendible hashing that makes it attractive is that the performance of the file does not degrade as the file grows, as opposed to static external hashing,

Dynamic Hashing: A precursor to extendible hashing was dynamic hashing proposed by Larson (1978), in which the addresses of the buckets were either the n high-order bits or n − 1 high-order bits, depending on the total number of keys belonging to the respective bucket. Dynamic hashing maintains a tree-structured directory with two types of nodes:

■ Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.

■ Leaf nodes—these hold a pointer to the actual bucket with records.

An example of the dynamic hashing appears in Figure 16.12. Four buckets are shown ("000", "001", "110", and "111") with high-order 3-bit addresses (corresponding to the global depth of 3), and two buckets ("01" and "10") are shown with high-order 2-bit addresses (corresponding to the local depth of 2). The latter two are the result of collapsing the "010" and "011" into "01" and collapsing "100" and "101" into "10".

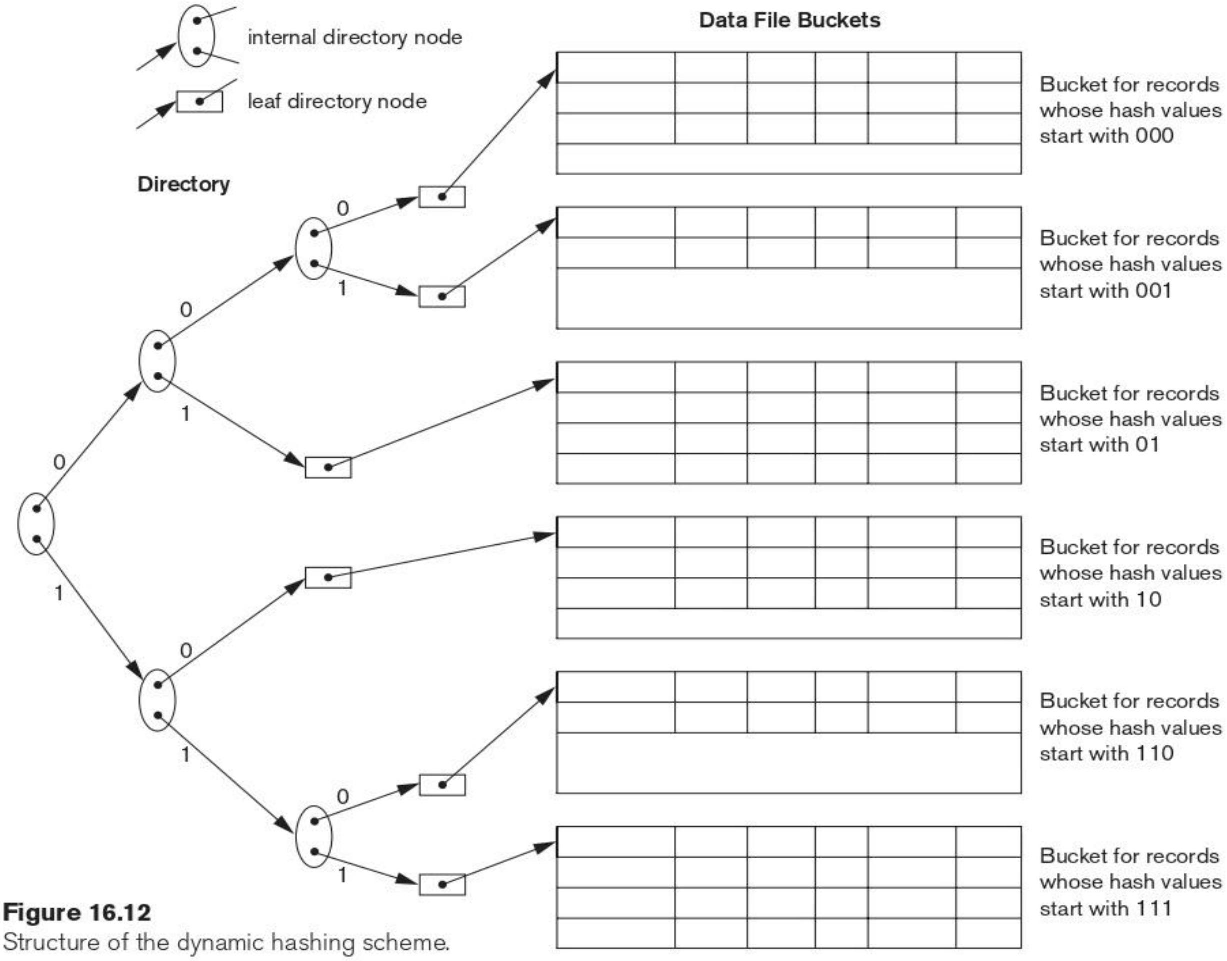

internal directory node
leaf directory node
Directory
Data File Buckets
0
1
0
1
0
1
0
1
0
1
Bucket for records whose hash values start with 000
Bucket for records whose hash values start with 001
Bucket for records whose hash values start with 01
Bucket for records whose hash values start with 10
Bucket for records whose hash values start with 110
Bucket for records whose hash values start with 111

Figure 16.12
Structure of the dynamic hashing scheme.

Linear Hashing: The idea behind linear hashing, proposed by Litwin (1980), is to allow a hash file to expand and shrink its number of buckets dynamically without needing a directory. Suppose that the file starts with M buckets numbered 0, 1, … , M − 1 and uses the mod hash function h(K) = K mod M; this hash function is called the **initial hash function h_i.**

When a collision leads to an overflow record in any file bucket, the first bucket in the file—bucket 0—is split into two buckets: the original bucket 0 and a new bucket M at the end of the file. The records originally in bucket 0 are distributed between the two buckets based on a different hashing function $h_{i+1}(K)$ = K mod 2M.

The main advantages of linear hashing are that it maintains the load factor fairly constantly while the file grows and shrinks, and it does not require a directory.

Algorithm: The Search Procedure for Linear Hashing

```
if n = 0
    then m ← hj (K) (*m is the hash value of record with hash key K*)
 else begin
        m ← hj (K);
        if m < n then m ← hj+1 (K)
       end;
search the bucket whose hash value is m (and its overflow, if any);
```

Parallelizing Disk Access Using RAID Technology

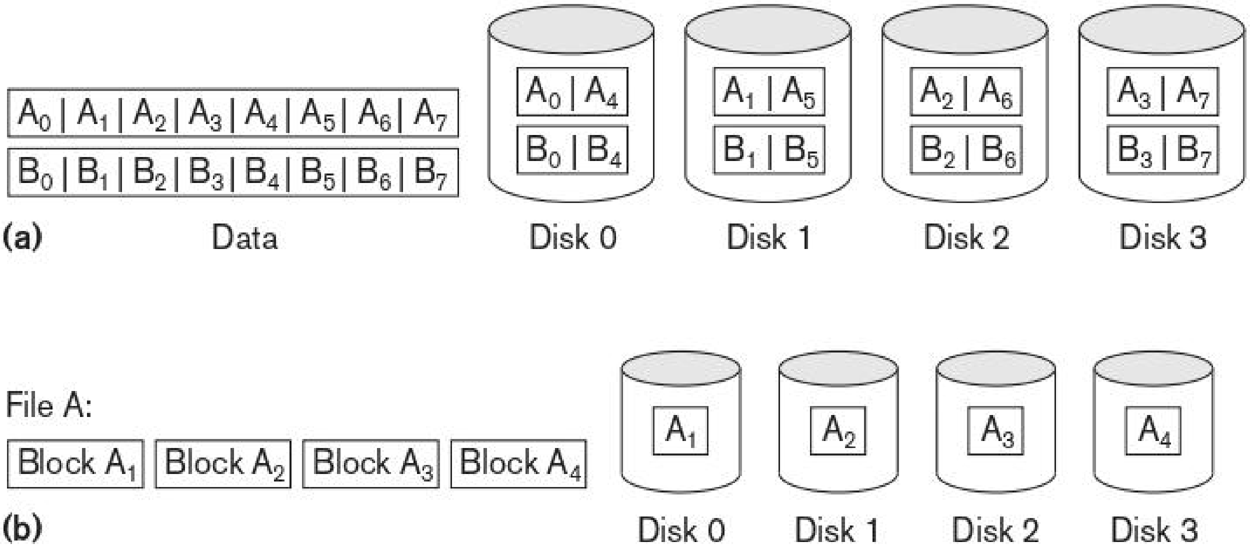
A major advance in secondary storage technology is represented by the development of RAID, which originally stood for redundant arrays of inexpensive disks. More recently, the I in RAID is said to stand for independent. The RAID idea received a very positive industry endorsement and has been developed into an elaborate set of alternative RAID architectures (RAID levels 0 through 6). The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.

The natural solution is a large array of small independent disks acting as a single higher performance logical disk. A concept called data striping is used, which utilizes parallelism to improve disk performance. **Data striping** distributes data transparently over multiple disks to make them appear as a single large, fast disk.

In **bit-level striping**, a byte is split and individual bits are stored on independent disks. Figure (a) illustrates bit-striping across four disks where the bits (0, 4) are assigned to disk 0, bits (1, 5) to disk 1, and so on.

Block-level striping stripes blocks across disks. It treats the array of disks as if it is one disk. Blocks are logically numbered from 0 in sequence. Disks in an m-disk array are numbered 0 to m – 1. With striping, block j goes to disk (j mod m). Figure (b) illustrates block striping with four disks (m = 4).

Figure: Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.



1. Improving Reliability with RAID

One technique for introducing redundancy is called mirroring or shadowing. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired.

Another solution to the problem of reliability is to store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure. The incorporation of redundancy must consider two problems: selecting a technique for computing the redundant information, and selecting a method of distributing the redundant information across the disk array.

The first problem is addressed by using error-correcting codes involving parity bits, or specialized codes such as Hamming codes. For the second problem, the two major approaches are either to store the redundant information on a small number of disks or to distribute it uniformly across all disks. The latter results in better load balancing. The different levels of RAID choose a combination of these options to implement redundancy and improve reliability.

2. Improving Performance with RAID

The disk arrays employ the technique of data striping to achieve higher transfer rates. Note that data can be read or written only one block at a time, so a typical transfer contains 512 to 8,192 bytes. **The bit-level data striping** consists of splitting a byte of data and writing bit j to the jth disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate.

The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to **block-level striping.** With block-level striping, multiple independent requests that access single blocks can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests.

3. RAID Organizations and Levels

Different RAID organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. In the initial proposal, levels 1 through 5 of RAID were proposed, and two additional levels—0 and 6—were added later.

RAID level 0 uses data striping, has no redundant data, and hence has the best write performance since updates do not have to be duplicated. It splits data evenly across two or more disks.

RAID level 1, which uses mirrored disks. In the latter, performance improvement is possible by scheduling a read request to the disk with shortest expected seek and rotational delay.

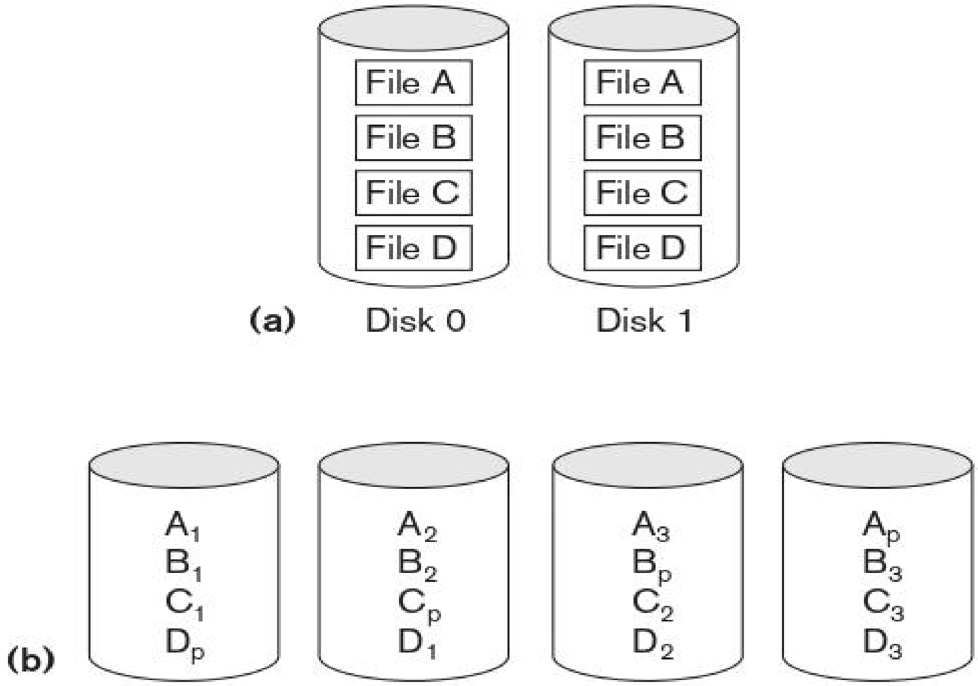
RAID level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components.

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed.

RAID Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Figure (b) shows an illustration of RAID level 5, where parity is shown with subscript p.

Finally**, RAID level 6** applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

Figure: Some popular levels of RAID. (a) RAID level 1: Mirroring of data on two disks. (b) RAID level 5: Striping of data with distributed parity across four disks.



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.

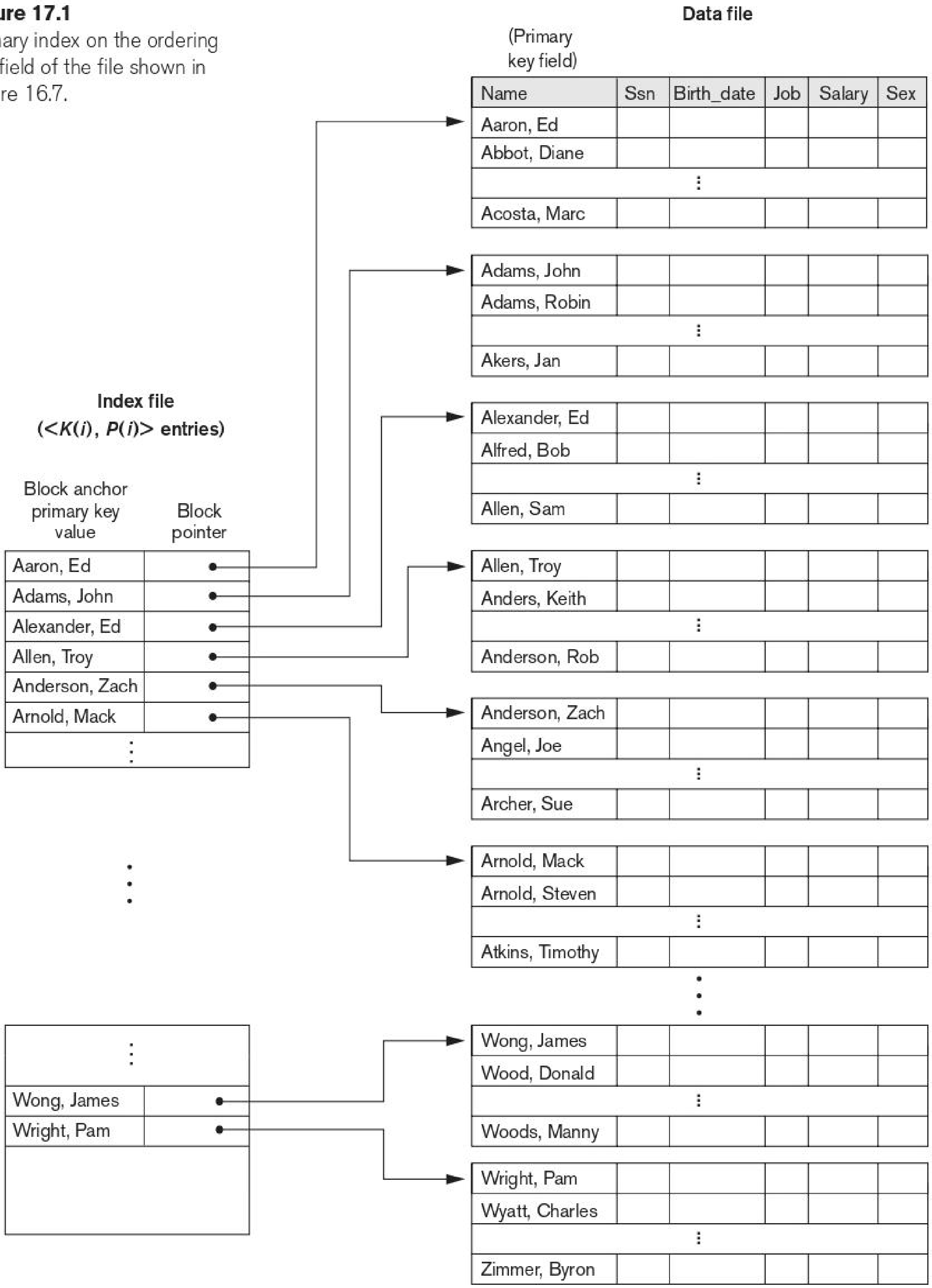
1. Primary Indexes

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.

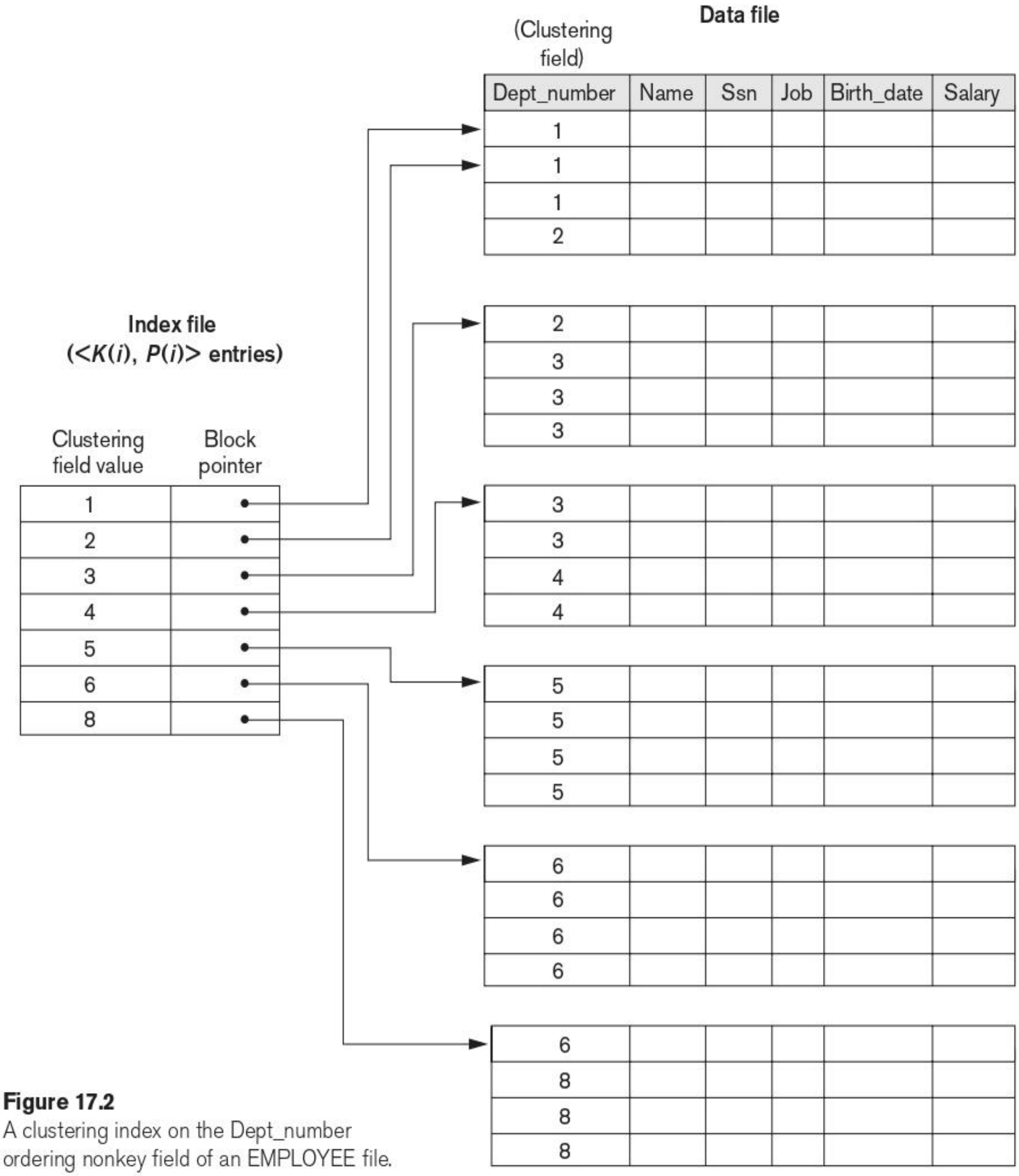
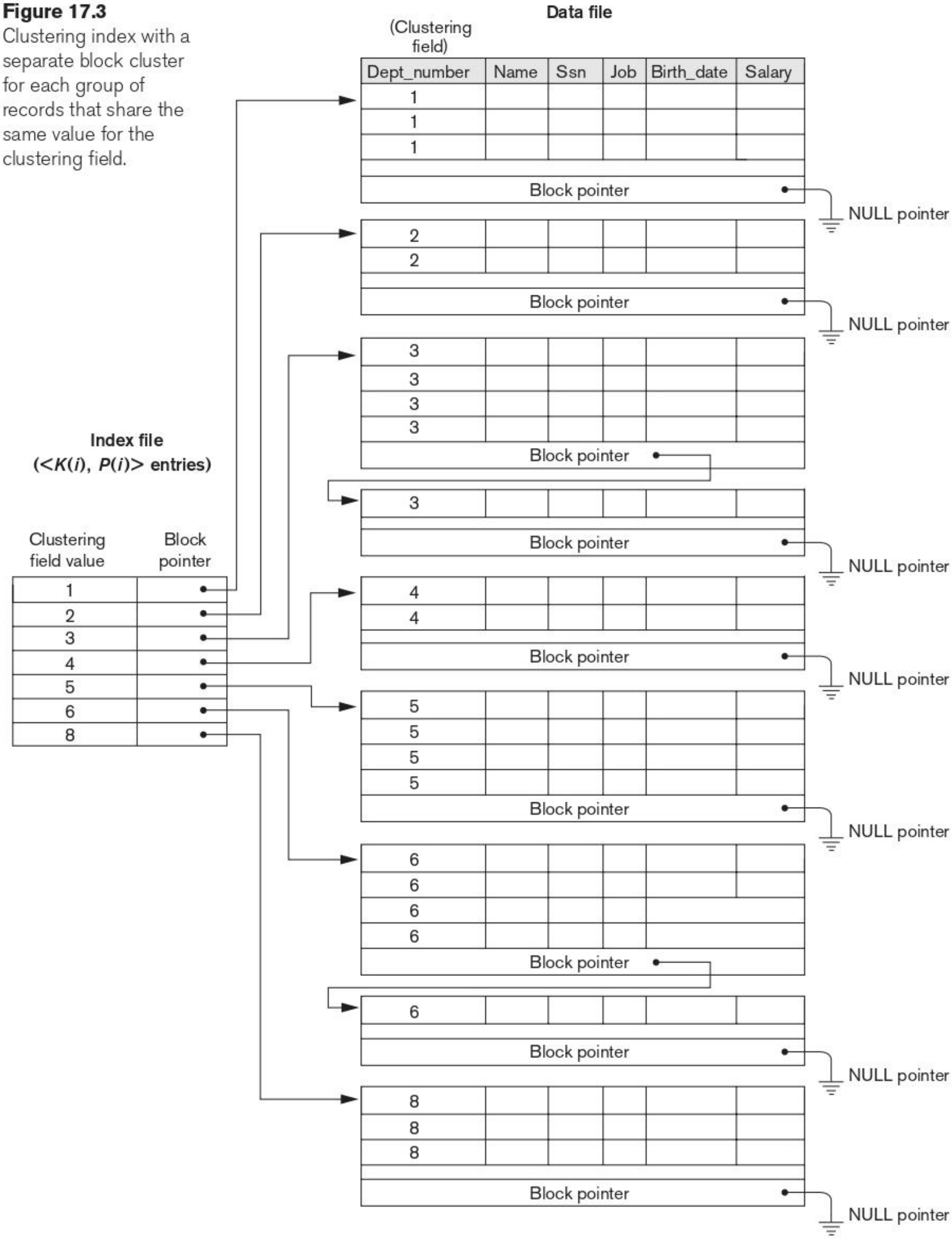


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

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.

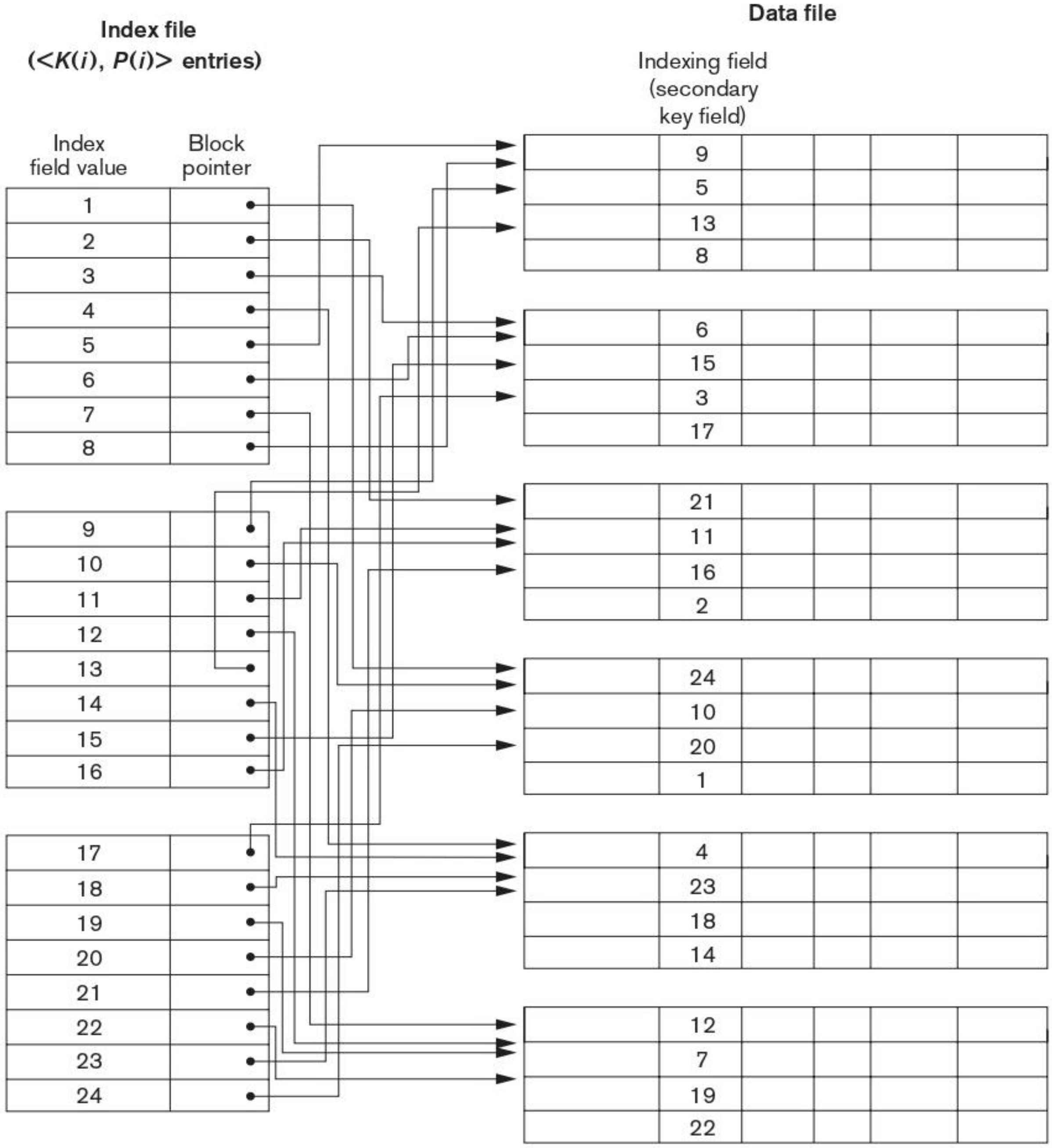
First we consider a secondary index access structure on a key (unique) field that has a distinct value for every record. Such a field is sometimes called a **secondary key**. A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number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



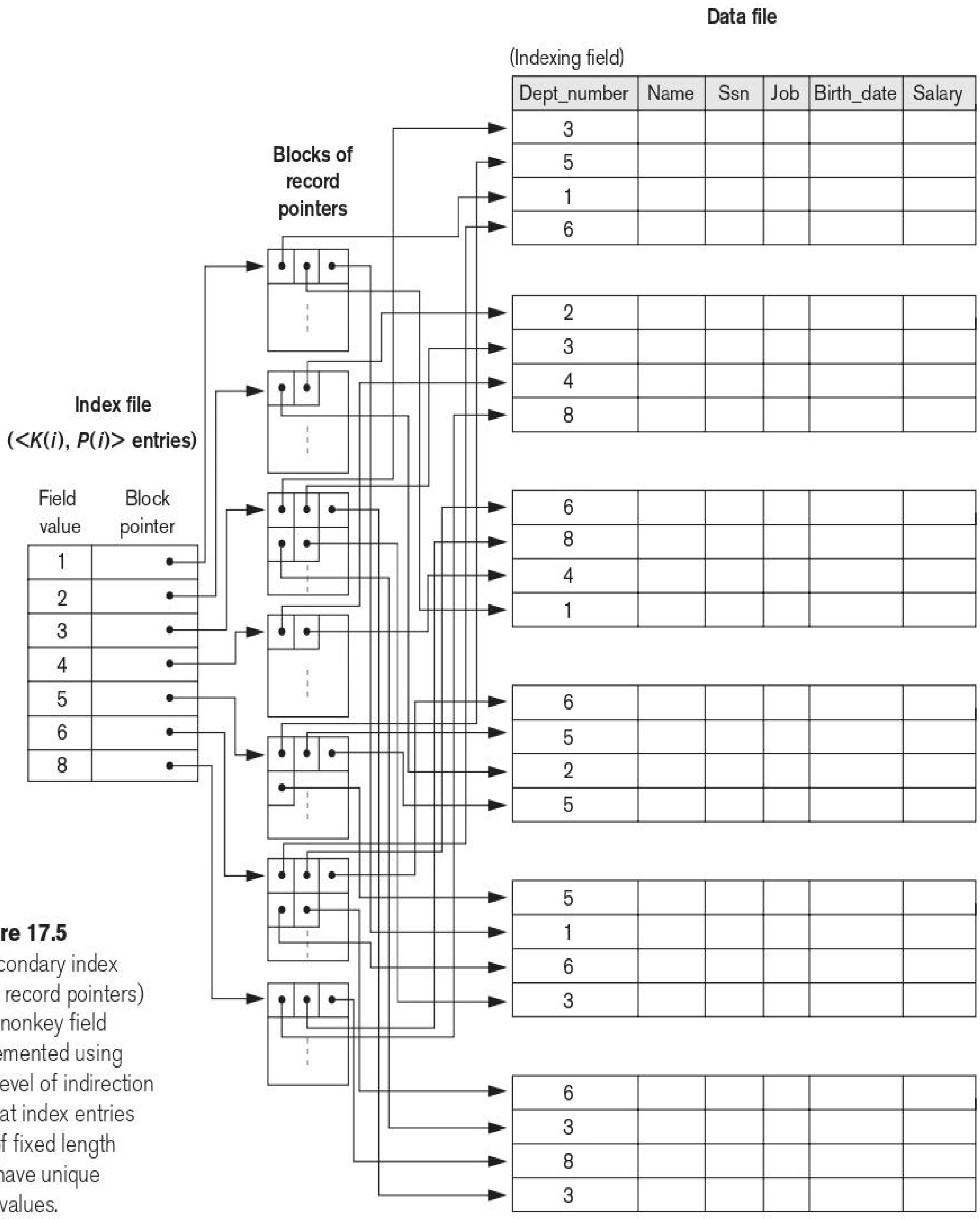


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

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.

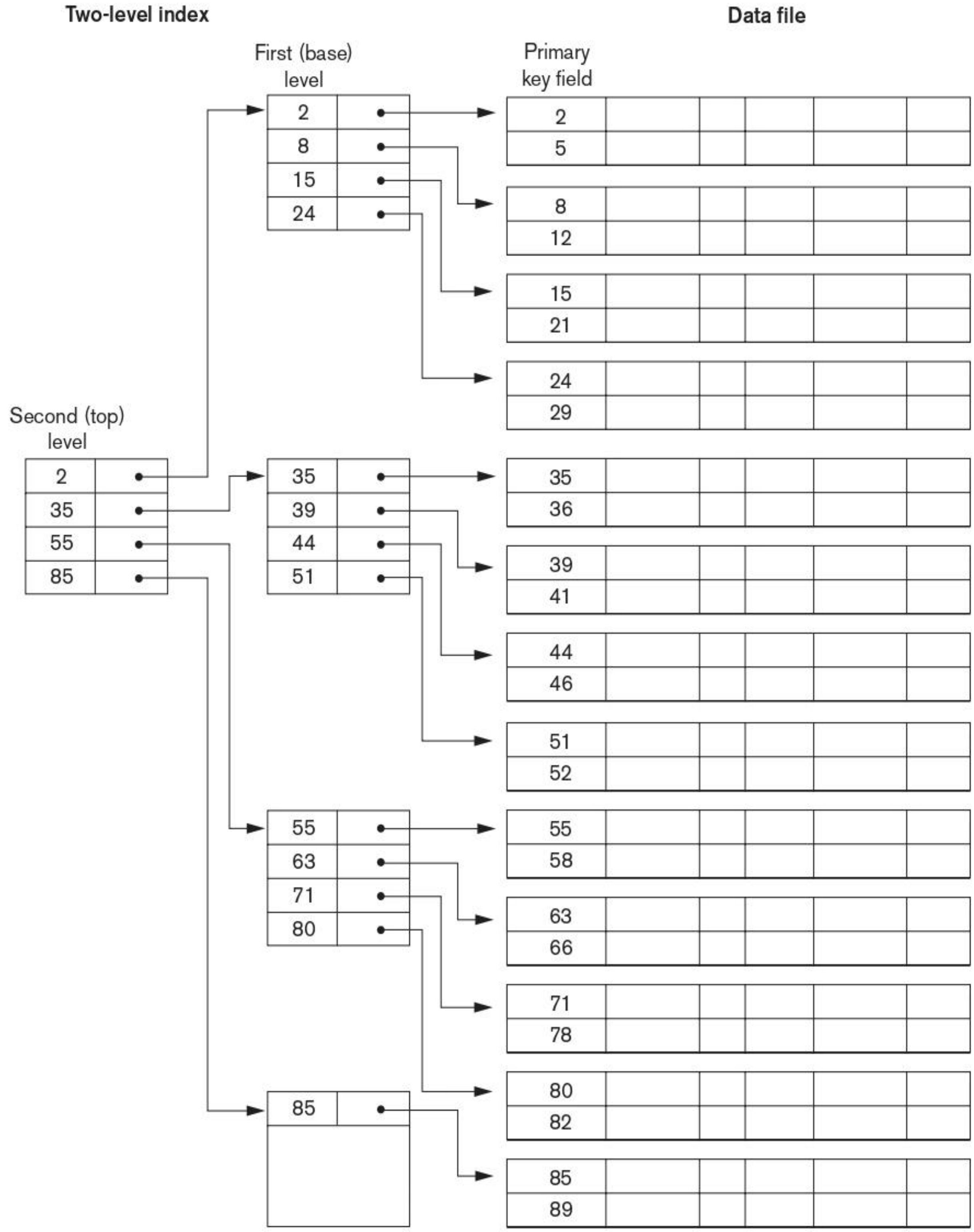
We can repeat this process for the second level. The third level, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is r3 = [(r2/fo)].

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.

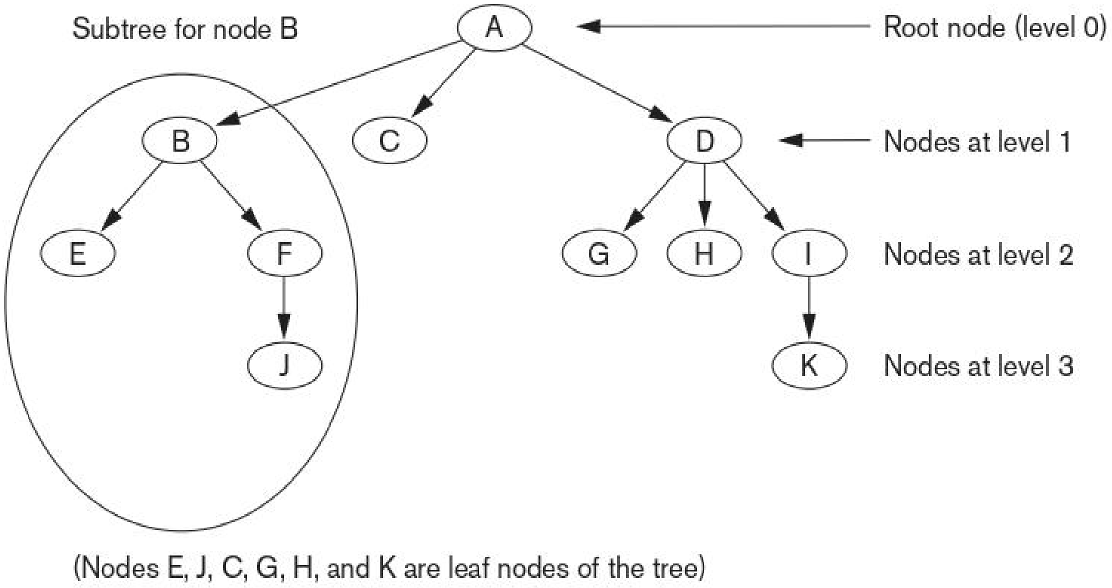
Dynamic Multilevel Indexes Using B-Trees and B+-Trees

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.

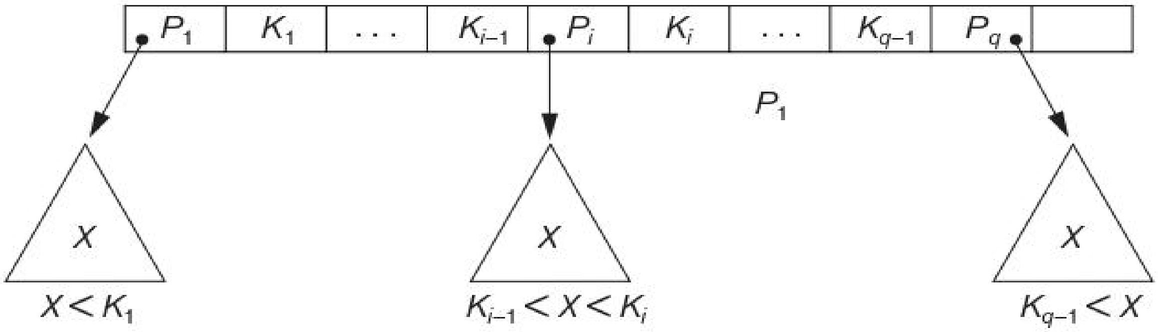
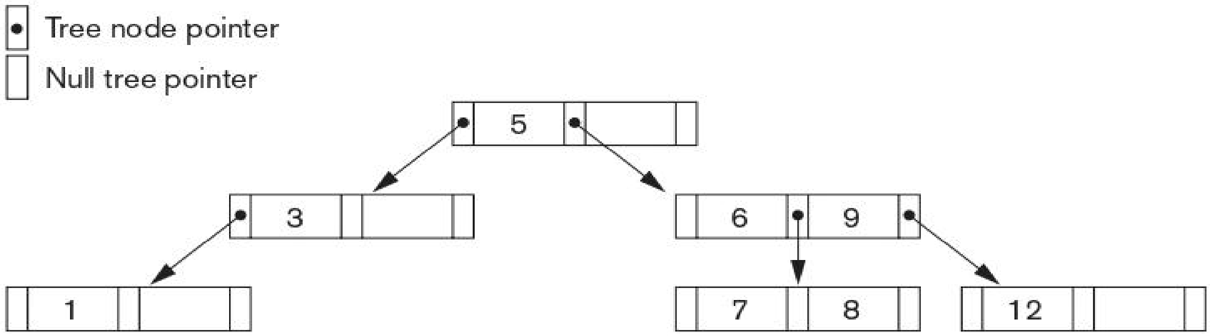


Figure 2: A search tree of order p = 3.



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
   $<P1, <K1,Pr1>, P2, <K2,Pr2>, \ldots , <Kq-1,Prq-1> , Pq>$

   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:

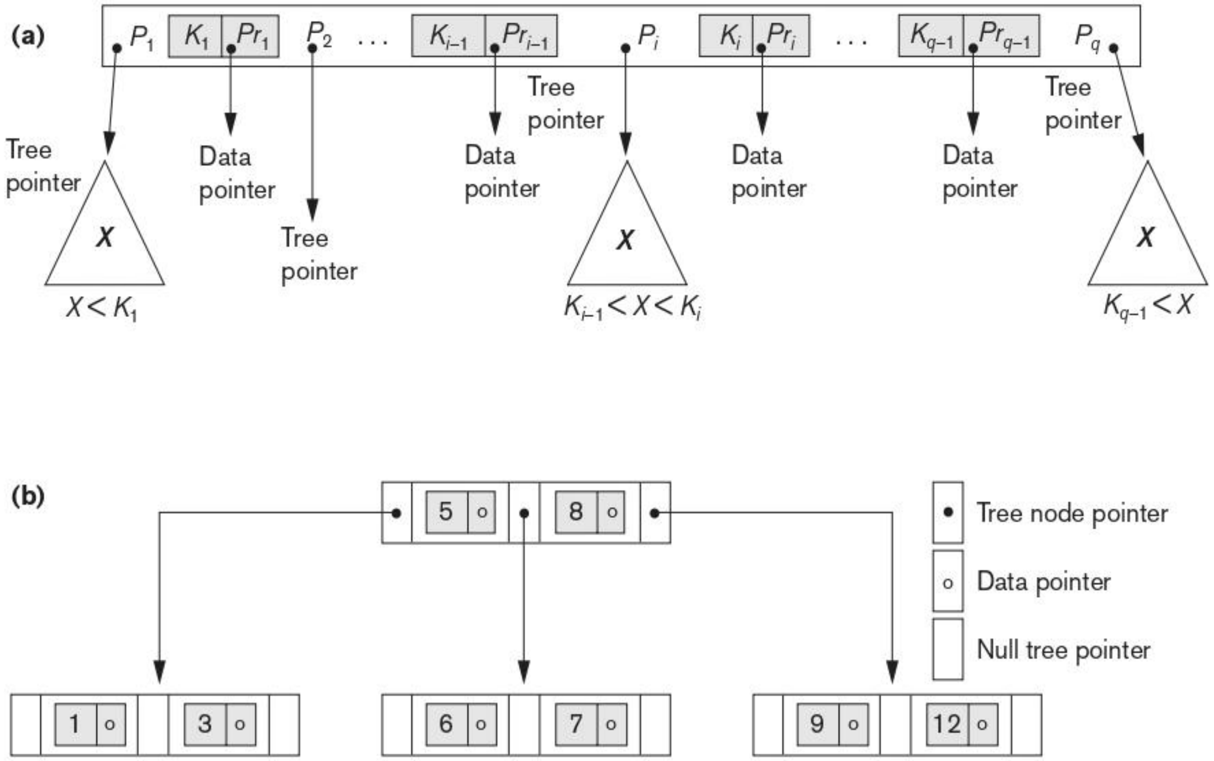
   $Ki-1 < X < Ki$ for $1 < i < q$; $X < Ki$ for $i = 1$; and $Ki-1 < X$ for $i = q$
4. Each node has at most p tree pointers.

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 $K_{i-1} < X \le K_i$ for $1 < i < q$; $X \le K_i$ 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

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

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

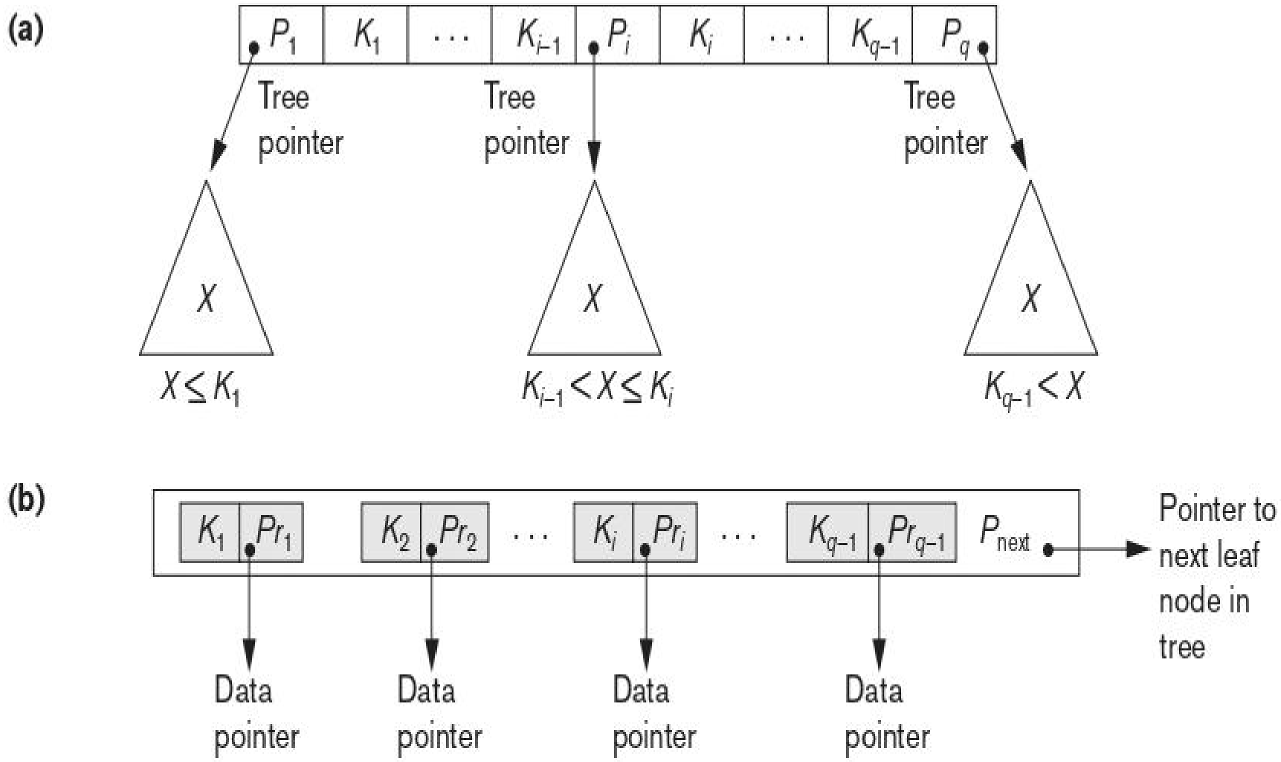
2. Within each leaf node, K1 ≤ K2 … , Kq−1, q ≤ p.

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.

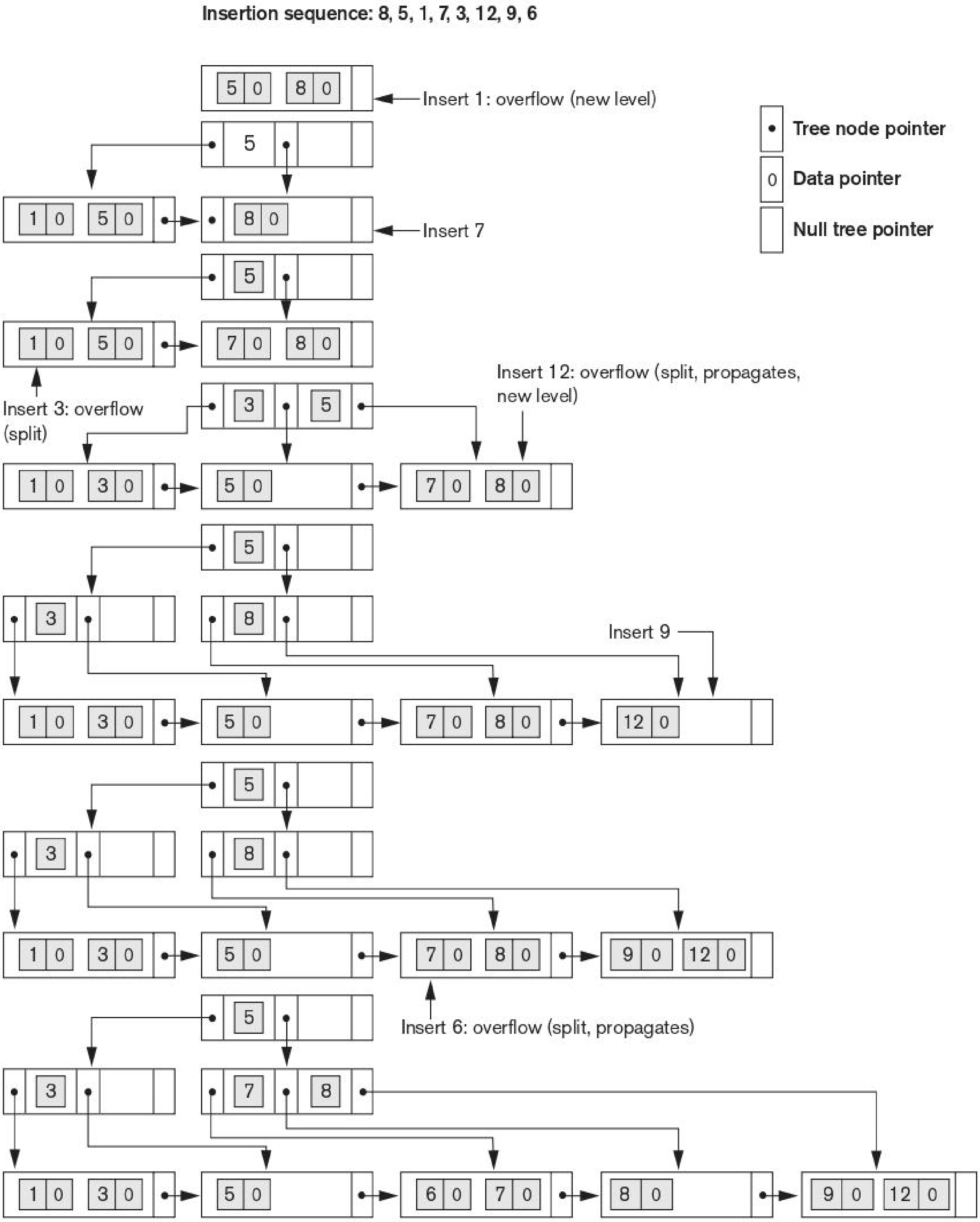
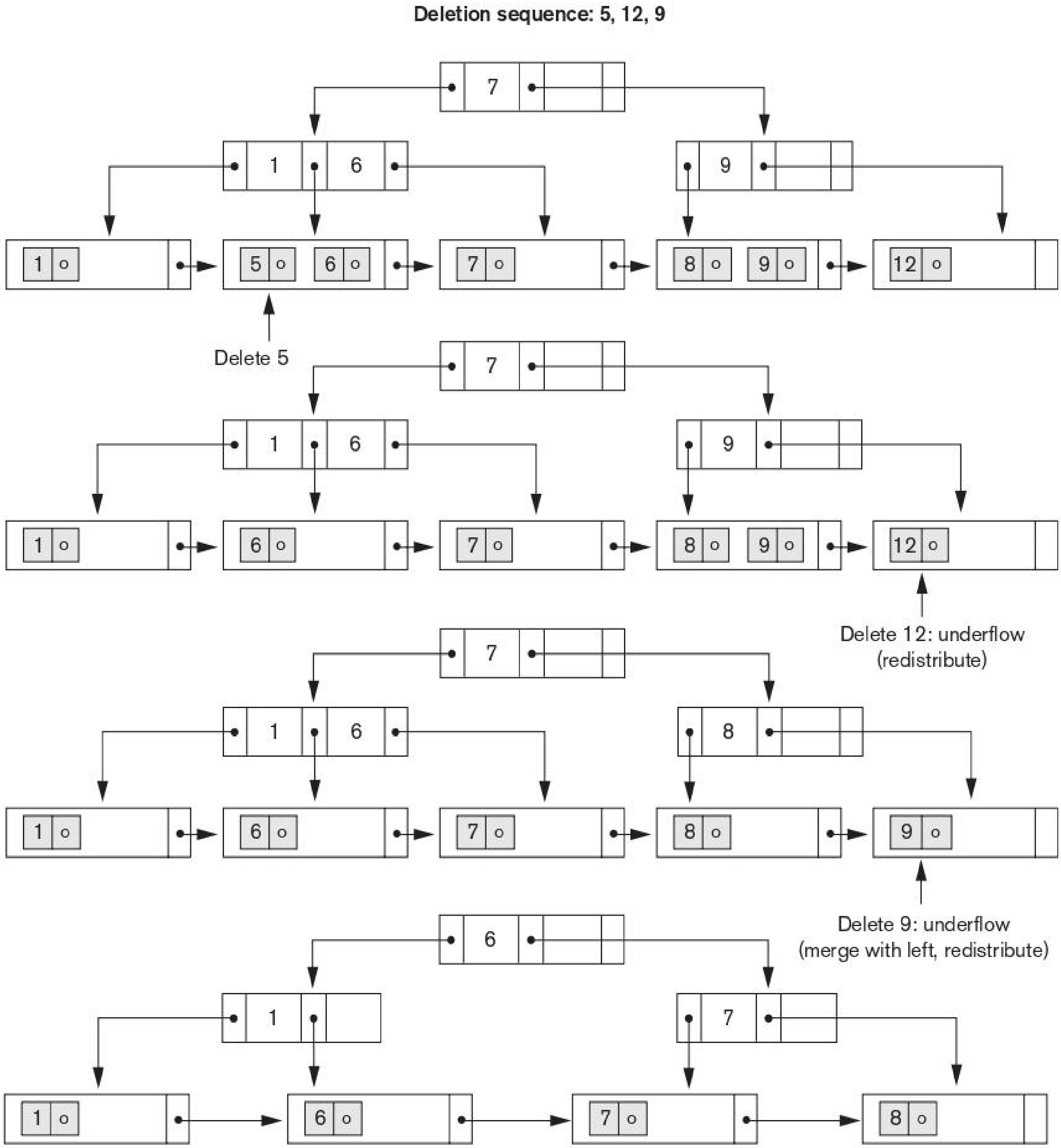

Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6
Insert 1: overflow (new level)
Insert 7
Insert 3: overflow (split)
Insert 12: overflow (split, propagates, new level)
Insert 9
Insert 6: overflow (split, propagates)
Tree node pointer
Data pointer
Null tree pointer
5
0
8
1
7
3
12
9
6

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.

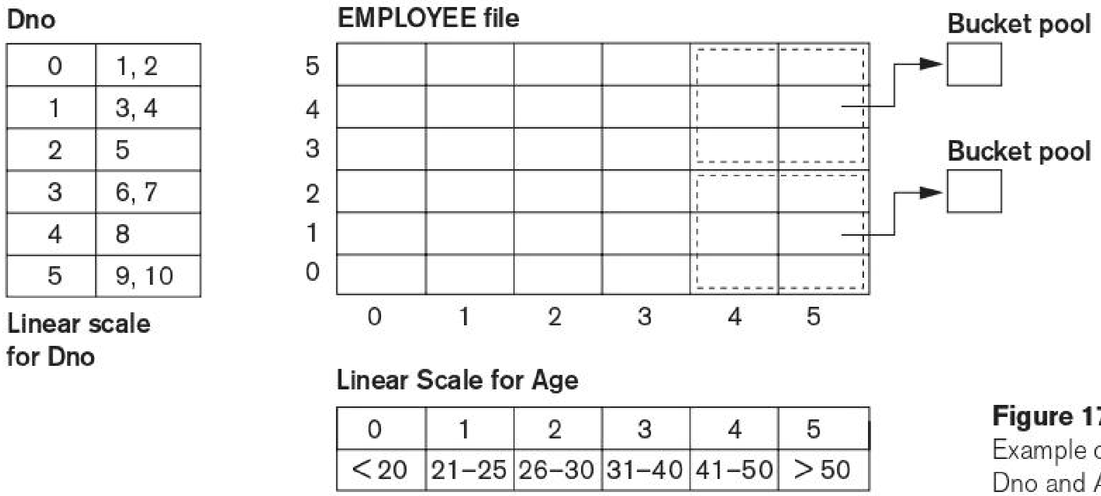
2. Partitioned Hashing:

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).



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- IV

Algorithms for query processing and Optimization

Translating SQL Queries into Relational Algebra

SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into ***query blocks***, which form the basic units that can be translated into the algebraic operators and optimized.

A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT

Consider the following SQL query on the EMPLOYEE relation:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT  MAX (Salary)
                  FROM EMPLOYEE
                 WHERE Dno=5 );
```

This query retrieves the names of employees who earn a salary that is greater than the ***highest salary in department 5.*** The query includes a nested subquery and hence would be decomposed into two blocks. The inner block is:

```
( SELECT MAX (Salary)
 FROM EMPLOYEE
 WHERE Dno=5 )
```

This retrieves the highest salary in department 5. The outer query block is:

```
 SELECT Lname, Fname
 FROM EMPLOYEE
WHERE Salary > c
```

Where c represents the result returned from the inner block.

The inner block could be translated into the following extended relational algebra expression:

$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$

and the outer block into the expression:

$\pi_{\text{Lname,Fname}}(\sigma_{\text{Salary>c}}(\text{EMPLOYEE}))$

The ***query optimizer*** would then choose an execution plan for each query block. It is more involved to optimize the more complex correlated nested subqueries, where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block. Many techniques are used in advanced DBMSs to unnest and optimize correlated nested subqueries.

Algorithms for SELECT and JOIN Operations

SELECT Operation

1. Implemention Options for the SELECT Operation

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions.

The following operations, specified on the relational database:

OP1: σSsn = ‘123456789’ (EMPLOYEE)
OP2: σDnumber > 5 (DEPARTMENT)
OP3: σDno= 5 (EMPLOYEE)
OP4: σDno= 5 AND Salary > 30000 AND Sex = ‘F’ (EMPLOYEE)
OP5: σEssn = ‘123456789’ AND Pno =10(WORKS_ON)
OP6: An SQL Query:

```
SELECT *
FROM EMPLOYEE
WHERE Dno IN (3,27, 49)
```

Search Methods for Simple Selection: A number of search algorithms are possible for selecting records from a file. These are also known as **file scans**, because they scan the records of a file to search for and retrieve records that satisfy a selection condition. If the search algorithm involves the use of an index, the index search is called an **index scan**. The following search methods (S1 through S7) are examples of some of the search algorithms that can be used to implement a select operation:

■ **S1—Linear search (brute force algorithm):** Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.

■ **S2—Binary search:**. If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search—which is more efficient than linear search—can be used.

■ **S3a—Using a primary index:** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = '123456789' in OP1—use the primary index to retrieve the record.

■ **S3b—Using a hash key:** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record.

■ **S4—Using a primary index to retrieve multiple records:** If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5); then retrieve all subsequent records in the (ordered) file.

■ **S5—Using a clustering index to retrieve multiple records:** If the selection condition involves an equality comparison on a nonkey attribute with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.

■ **S6—Using a secondary (B+-tree) index on an equality comparison:** This search method can be used to retrieve a single record if the indexing field is a **key** or to retrieve multiple records if the indexing field is **not a key**. Queries involving a range of values (e.g., 3,000 <= Salary <= 4,000) in their selection are called **range queries**. In case of range queries, the B+-tree index leaf nodes contain the indexing field value in order—so a sequence of them is used corresponding to the requested range of that field and provide record pointers to the qualifying records.

■ **S7a—Using a bitmap index:** If the selection condition involves a set of values for an attribute (e.g., Dnumber in (3,27,49) in OP6), the corresponding bitmaps for each value can be OR-ed to give the set of record ids that qualify.

■ **S7b—Using a functional index:** If there is a functional index defined as:

CREATE INDEX income_ix
ON EMPLOYEE (Salary + (Salary*Commission_pct));

then this index can be used to retrieve employee records that qualify.

2. Search Methods for Conjunctive Selection

If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

■ **S8—Conjunctive selection using an individual index:** If an attribute involved in any **single simple condition** in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive select condition.

■ **S9—Conjunctive selection using a composite index**: If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index exists on the combined fields

■ **S10—Conjunctive selection by intersection of record pointers:** If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the set of record pointers that satisfy the individual condition.

3. Search Methods for Disjunctive Selection:

Compared to a conjunctive selection condition, a **disjunctive condition** is much harder to process and optimize. For example, consider OP4′:

OP4′: $\sigma_{\text{Dno=5 OR Salary > 30000 OR Sex ='F'}}$ (EMPLOYEE)

With such a condition, the records satisfying the disjunctive condition are the **union** of the records satisfying the individual conditions. Hence, if **any one** of the conditions does not have an access path, we are compelled to use the brute force, linear search approach. Only if an access path exists on every simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the union operation to eliminate duplicates.

Implementing the JOIN Operation

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN.

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins**. The number of possible ways to execute multiway joins grows rapidly because of the combinatorial explosion of possible join orderings.

The algorithms for a join operation of the form:

$$R \bowtie_{A=B} S$$

where A and B are the **join attributes**, which should be domain-compatible attributes of R and S, respectively. The most common techniques for performing a join, using the following sample operations:

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT
OP7: DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

1. Methods for Implementing Joins

■ **J1—Nested-loop join (or nested-block join):** This is the default (brute force) algorithm because it does not require any special access paths on either file in the join. For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].

■ **J2—Index-based nested-loop join (using an access structure to retrieve the matching records):** If an index (or hash key) exists for one of the two join attributes—say, attribute B of file S—retrieve each record t in R (loop over file R), and then use the access structure to retrieve directly all matching records s from S that satisfy s[B] = t[A].

■ **J3—Sort-merge join:** If the records of R and S are physically sorted (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they may be sorted first by using external sorting.

■ **J4—Partition-hash join (or just hash-join):** The records of files R and S are partitioned into smaller files. The partitioning of each file is done using the same hashing function **h** on the join attribute A of R (for partitioning file R) and B of S (for partitioning file S). First, a single pass through the file with fewer records (say, R) hashes its records to the various partitions of R; this is called the **partitioning phase.** The collection of records with the same value of h(A) are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase, called the **probing phase**, a single pass through the other file (S) then hashes each of its records using the same hash function h(B) to probe the appropriate bucket, and that record is combined with all matching records from R in that bucket.

2. How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join

The buffer space available has an important effect on some of the join algorithms. The algorithm can read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer-loop blocks that are currently in main memory for matching records. This reduces the total number of block accesses. An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result— whenever it is filled. This result buffer block then is reused to hold additional join result records.

In the nested-loop join, it makes a difference which file is chosen for the outer loop and which for the inner loop. If EMPLOYEE is used for the outer loop, each block of EMPLOYEE is read once, and the entire DEPARTMENT file (each of its blocks) is read once for each time we read in blocks of the EMPLOYEE file.

The join algorithm uses a buffer to hold the joined records of the result file. Once the buffer is filled, it is written to disk and its contents are appended to the result file, and then refilled with join result records.

3. How the Join Selection Factor Affects Join Performance:

Another factor that affects the performance of a join, particularly the single-loop method J2, is the fraction of records in one file that will be joined with records in the other file. We call this the join selection factor of a file with respect to an equijoin condition with another file. This factor depends on the particular equijoin condition between the two files.

4. General Case for Partition-Hash Join

The hash-join method J4 is also efficient. In this case, only a single pass is made through each file, whether or not the files are ordered. If the hash table for the smaller of the two files can be kept entirely in main memory after hashing (partitioning) on its join attribute, the implementation is straightforward. If, however, the partitions of both files must be stored on disk, the method becomes more complex, and a number of variations to improve the efficiency have been proposed. We discuss two techniques: the general case of partition-hash join and a variation called hybrid hash-join algorithm, which has been shown to be efficient.

In the general case of **partition-hash join**, each file is first partitioned into M partitions using the same **partitioning hash function** on the join attributes. Then, each pair of corresponding partitions is joined. For example, suppose we are joining relations R and S on the join attributes R.A and S.B:

$$R \bowtie_{A=B} S$$

In the partitioning phase, R is partitioned into the M partitions R_1, R_2, … , R_M, and S into the M partitions S_1, S_2, …, S_M.

5. **Hybrid Hash-Join**

The **hybrid hash-join algorithm** is a variation of partition hash-join, where the joining phase for one of the partitions is included in the partitioning phase. To illustrate this, let us assume that the size of a memory buffer is one disk block; that n_B such buffers are available; and that the partitioning hash function used is h(K) = K mod M, so that M partitions are being created, where $M < n_B$.

Algorithms for PROJECT and SET Operations

A PROJECT operation $\pi_{<\text{attribute list}>}(R)$ from relational algebra implies that after projecting R on only the columns in the list of attributes, any duplicates are removed by treating the result strictly as a set of tuples. However, the SQL query:

SELECT Salary
FROM EMPLOYEE

produces a list of salaries of all employees. If there are 10,000 employees and only 80 distinct values for salary, it produces a one column result with 10,000 tuples. This operation is done by simple linear search by making a complete pass through the table.

Getting the true effect of the relational algebra $\pi_{<\text{attribute list}>}(R)$ operator is straightforward to implement if <attribute list> includes a key of relation R, because in this case the result of the operation will have the same number of tuples as R, but with only the values for the attributes in <attribute list> in each tuple. If <attribute list> does not include a key of R, ***duplicate tuples must be eliminated***. This can be done by sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting. Hashing can also be used to eliminate duplicates: as each record is hashed and inserted into a bucket of the hash file in memory

A sketch of the algorithm is:

Algorithm: Implementing the operation T ← $\pi_{<\text{attribute list}>}$ (R).

```
create a tuple t[] in T ′ for each tuple t in R;
    (* T′ contains the projection results before duplicate elimination *)
If <attribute list>  includes a key of R
    then T ← T ′

else { sort the tuples in T ′;
     set i ← 1, j ← 2;
     while i ≤ n
     do { output the tuple T′[i] to T;
            while T ′[i] = T′[j] and j ≤ n do j ← j + 1;   (* eliminate duplicates *)
            i ← j; j ← i + 1
        }
  }
 (*T contains the projection result after duplicate elimination*)
```

Set operations UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT—are sometimes expensive to implement, since UNION, INTERSECTION, MINUS or SET DIFFERENCE are set operators and must always return distinct results.

In particular, the CARTESIAN PRODUCT operation R × S is expensive because its result includes a record for each combination of records from R and S. Also, each record in the result includes all attributes of R and S. The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE—apply only to **type-compatible** (or union-compatible) relations, which have the same number of attributes and the same attribute domains.

Algorithm: Implementing the operation T ← R ∪ S.

```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and ( j ≤ m)
  do { if  R(i) > S(j)
           then { output S(j) to T;
                     set j ← j + 1
                 }
        elseif R(i) < S(j)
             then {  output R(i) to T;
                        set i ← i + 1
```

```
                    }
            else set j ← j + 1      (* R(i)=S(j), so we skip one of the duplicate tuples *)
    }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
    if (j ≤ m) then add tuples S(j) to S(m) to T;
```

Algorithm: Implementing the operation T ← R ∩ S.

```
     sort the tuples in R and S using the same unique sort attributes;
     set i ← 1, j ← 1;
     while (i ≤ n) and (j ≤ m)
     do { if R(i) > S(j)
                then set j ← j + 1
            elseif R(i) < S(j)
                  then set i ← i + 1
            else {  output R(j) to T;          (* R(i) = S( j), so we output the tuple *)
                     set i ← i + 1, j ← j + 1
             }
       }
```

Algorithm: Implementing the operation T ← R – S.

```
    sort the tuples in R and S using the same unique sort attributes;
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m)
    do { if R(i) > S(j)
                then set j ← j + 1
           elseif R(i) < S(j)
                then {   output R(i) to T; (* R(i) has no matching S( j), so output R(i) *)
                          set i ← i + 1
                     }
           else set i ← i + 1, j ← j + 1
    }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
```

Hashing can also be used to implement UNION, INTERSECTION, and SET DIFFERENCE. One table is first scanned and then partitioned into an in-memory hash table with buckets, and the records in the other table are then scanned one at a time and used to probe the appropriate partition.

Introduction to Transaction Processing Concepts and Theory

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.

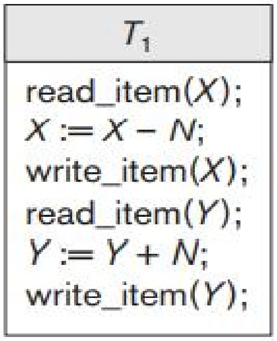
3. **Buffers:** The DBMS will generally generally maintain a number of buffers in main memory that hold database disk block containing the database items being processed.

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.

3. Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

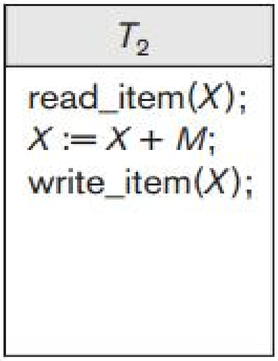
For example, the fig, shows transaction T_1:



T_1
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

A transaction T_1 that transfers N reservations from one flight stored in the database item named X to another flight in the database item named Y.

The following fig. shows another transaction T_2:



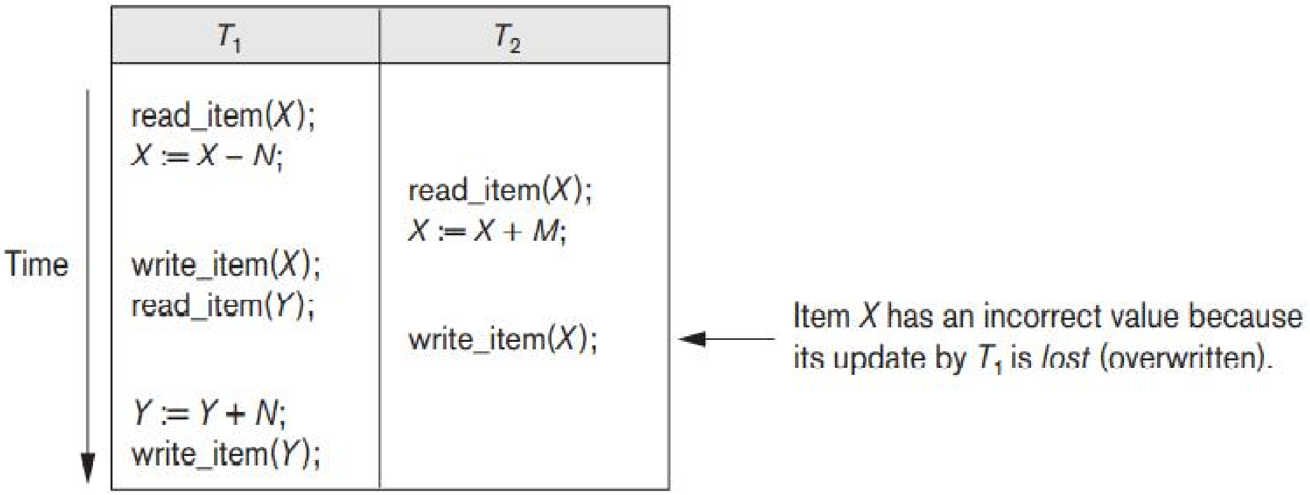
T_2
read_item(X);
$X := X + M$;
write_item(X);

A simpler transaction T_2 reserves M seats on the first flight (X) referenced in transaction T_1.

The types of problems may encounter with these two run concurrently.

a. **The Lost Update Problem**:

This problem occurs when two transactions that access the same database items have their operations are interleaved that makes the value of some database items are incorrect.



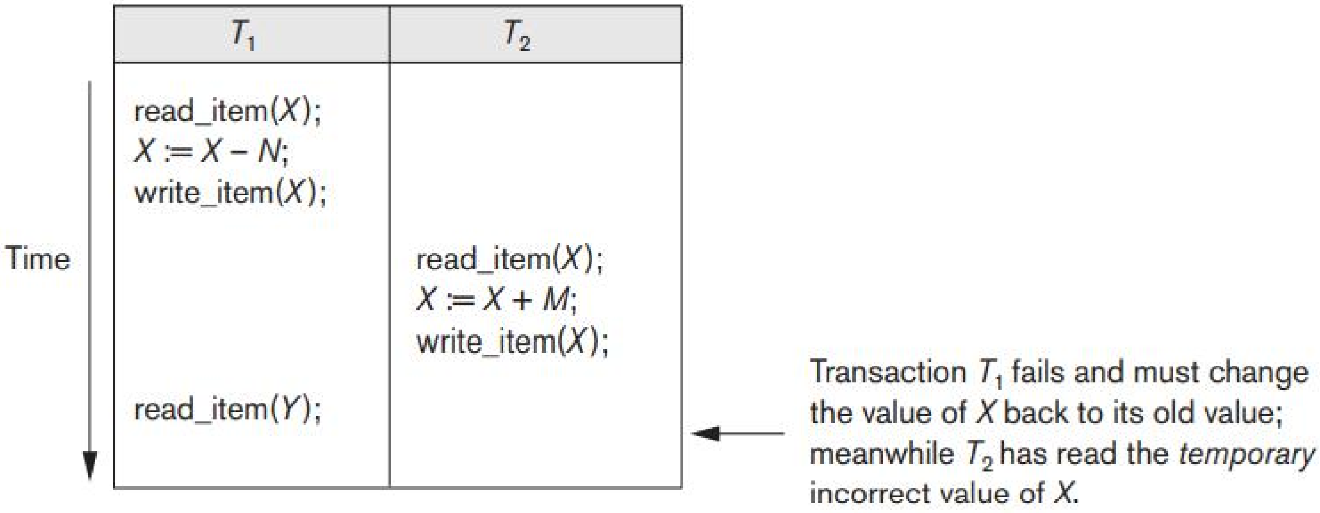
T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

For example, if X = 80 at the start N = 5 and M = 4 the final result X = 79. But in the interleaving of operations shown in Figure it is X = 84 because the update in T1 that removed the five seats from X was lost.

b. The Temporary Update (or Dirty Read) Problem:

This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed (read) by another transaction before it is changed back to its original value.



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);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

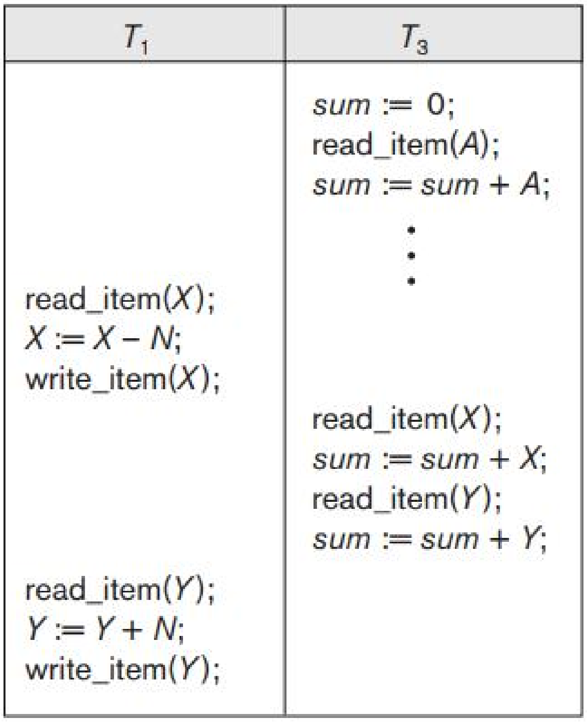
The transaction T1 updates item X and then fails before completion, so the system must roll back X to its original value. Before it can do, transaction T2 reads the temporary value of X.

The value of item X that is read by T2 is called **dirty data** because it has been created by a transaction that has not completed and committed. This problem is also known as the **dirty read problem**.

c. The Incorrect Summary Problem.

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Ex:



T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	⋮

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

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

T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

The transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing.

The result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

4. Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for either all the operations in the transaction are completed and their effect is recorded permanently in the database, or the transaction does not effect on the database.

Types of Failures: Failures are generally classified as transaction, system, and media failures.

1. **A computer failure (system crash):** A **hardware, software, or network error** occurs in the computer system during transaction execution.

2. **A transaction or system error:** Some operation in the transaction may cause it to fail, such as **integer overflow or division by zero**.

3. **Local errors:** During the transaction execution, certain conditions may occur that **cancellation of the transaction**. For example, Insufficient account balance in a banking database, a transaction, such as a fund withdrawal, to be canceled.

4. **Concurrency control:** The concurrency control method may abort a transaction because it **violates serializability.**

5. **Disk failure:** Some disk blocks may lose their data because of a **read or write head crash.**

6. **Physical problems:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. The failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

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.

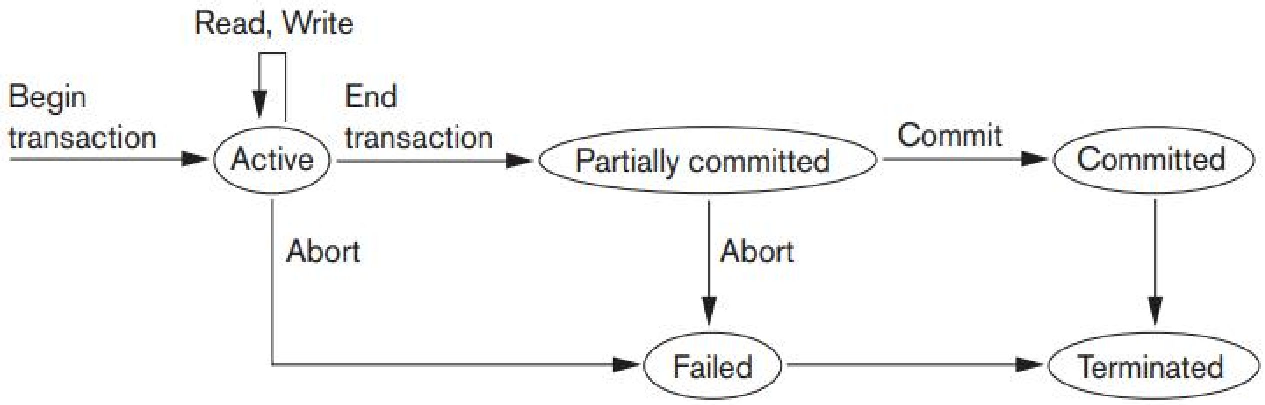
■ **READ or WRITE**: These specify **read or write operations** on the database items.

■ **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.

Characterizing Schedules

Characterizing Schedules Based on Recoverability

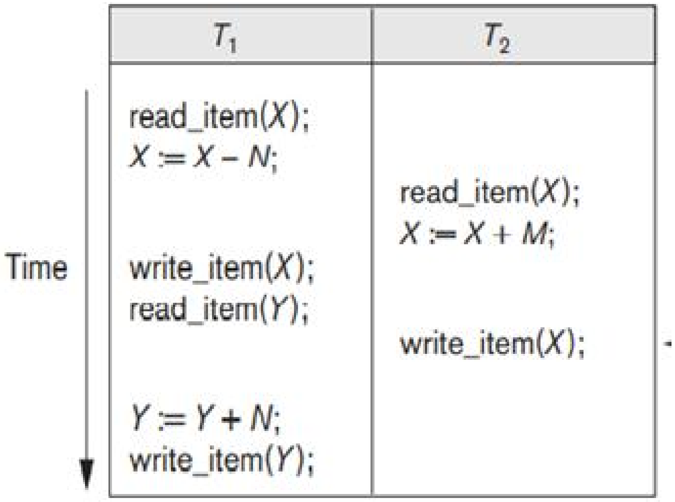
The order of execution of operations from all the various transactions is known as a **schedule (**or **history**).

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:



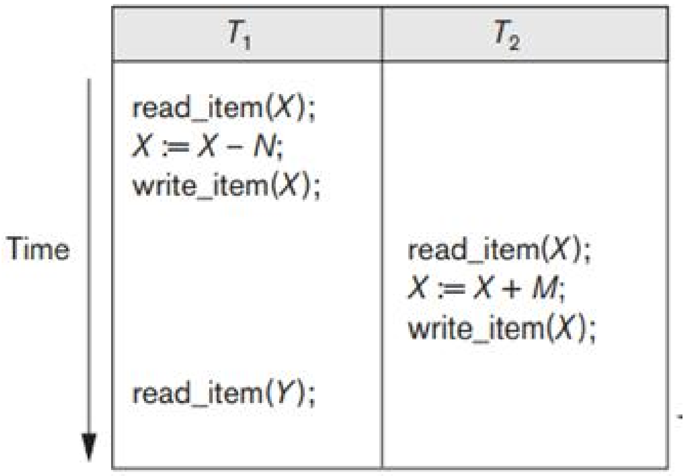
T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

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.

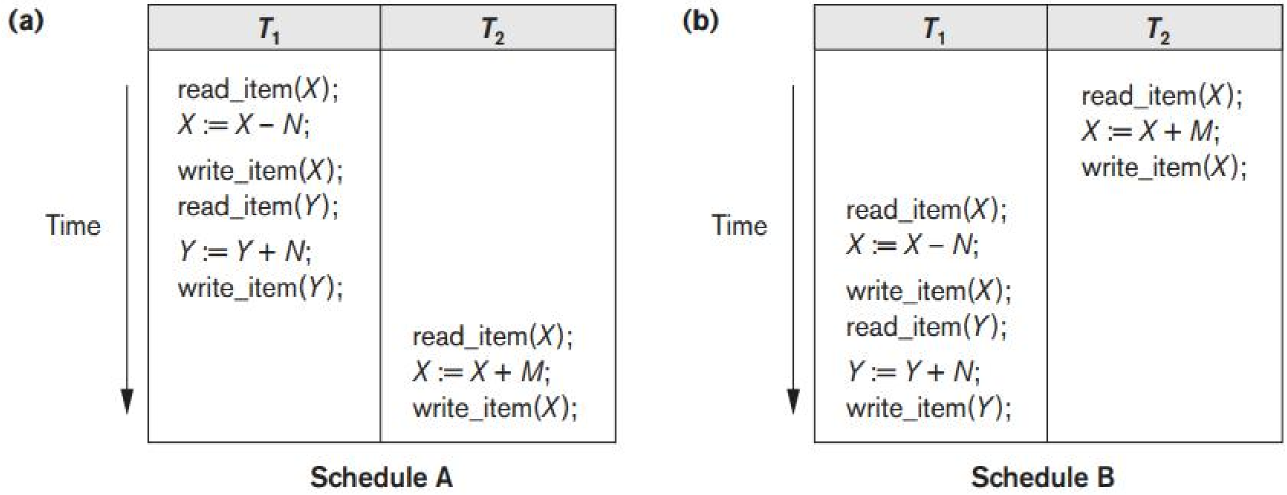
In a recoverable schedule, no committed transaction needs to be rolled back. However, this phenomenon known as cascading rollback to occur, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.

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.



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

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.

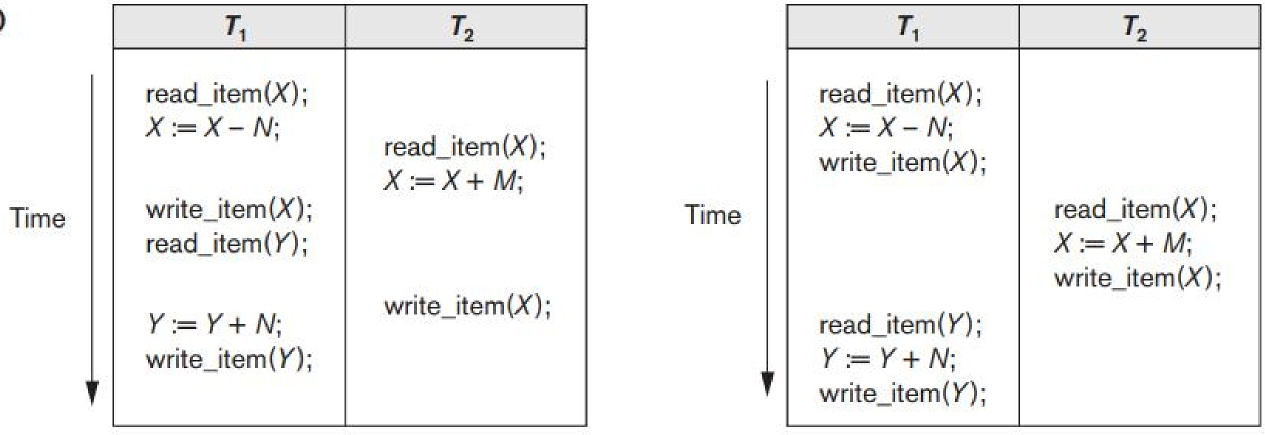
In serial schedule every transaction is executed from beginning to end without any interface from the operations of others transactions, we get a correct end result.

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:

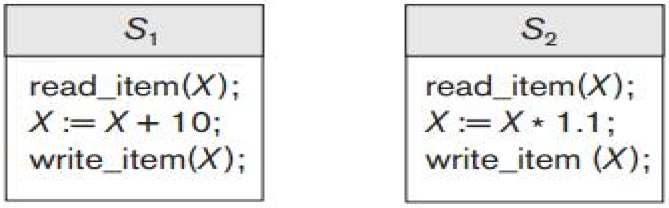
Assume that the initial values of database items are X = 90 and Y = 90 and N = 3 and M = 2.

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	S_2

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

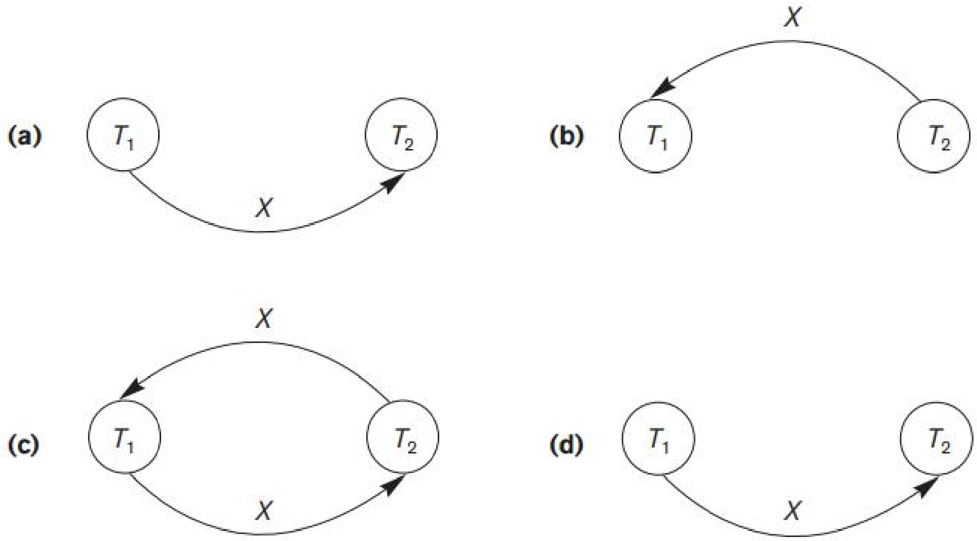
Algorithm: Testing Conflict Serializability of a Schedule S

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.

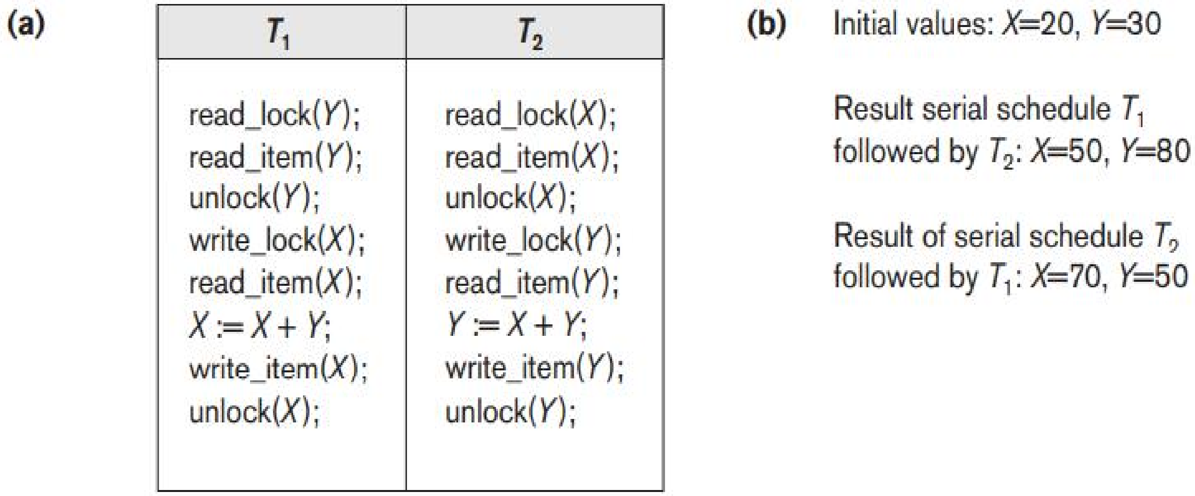
For example, a transaction T issue a read_lock(X) and then later to **upgrade the lock** by issuing a write_lock(X) operation.

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

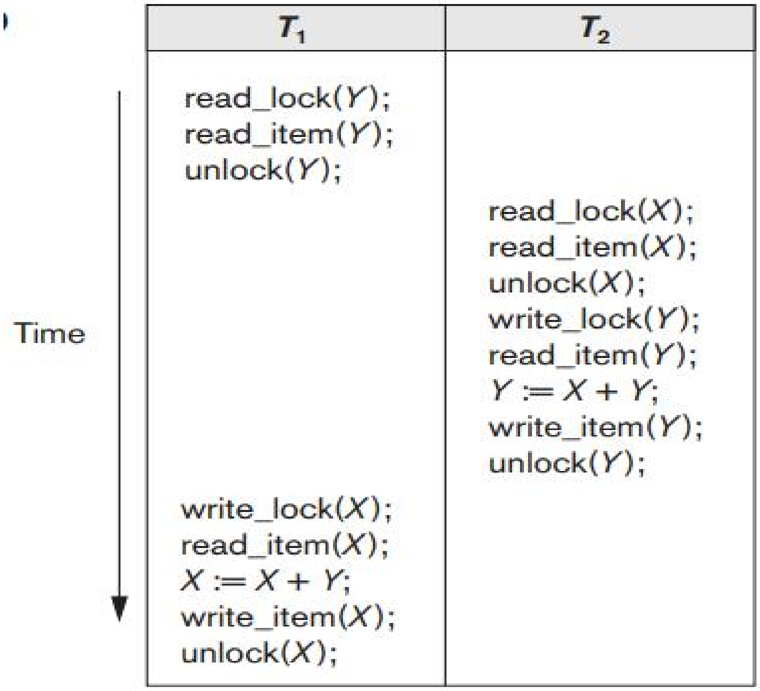
(b) Initial values: X=20, Y=30

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.

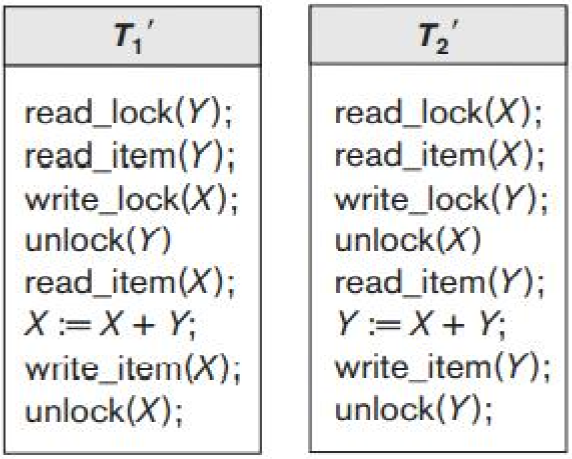
In the two phase locking a transaction can be divided into 2 phases. They are **growing phase, shinking phase.**

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.

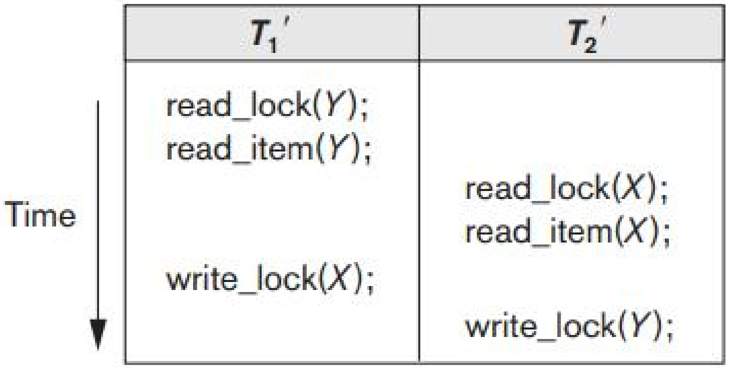
Rigorous 2Phase Locking: In this a transaction T does not release any of its locks (read lock or write lock) until after it commits or aborts.

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.

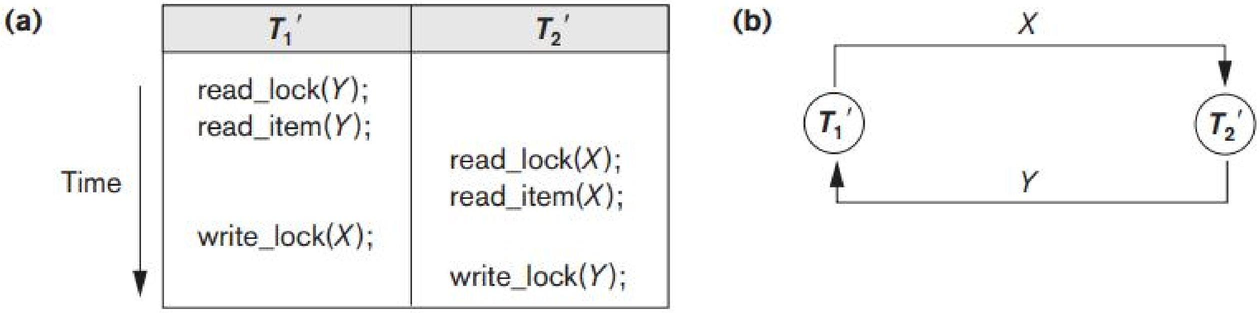
Deadlock Detection:

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:

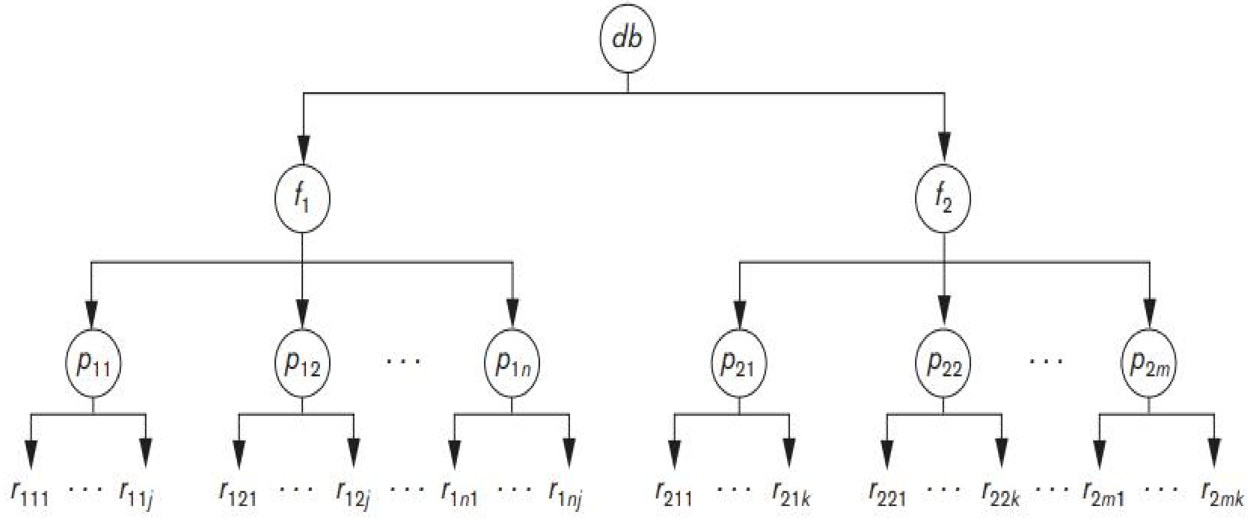
- ■ A database record
- ■ A field value of a database record
- ■ A disk block
- ■ A whole file
- ■ The whole database

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.



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).

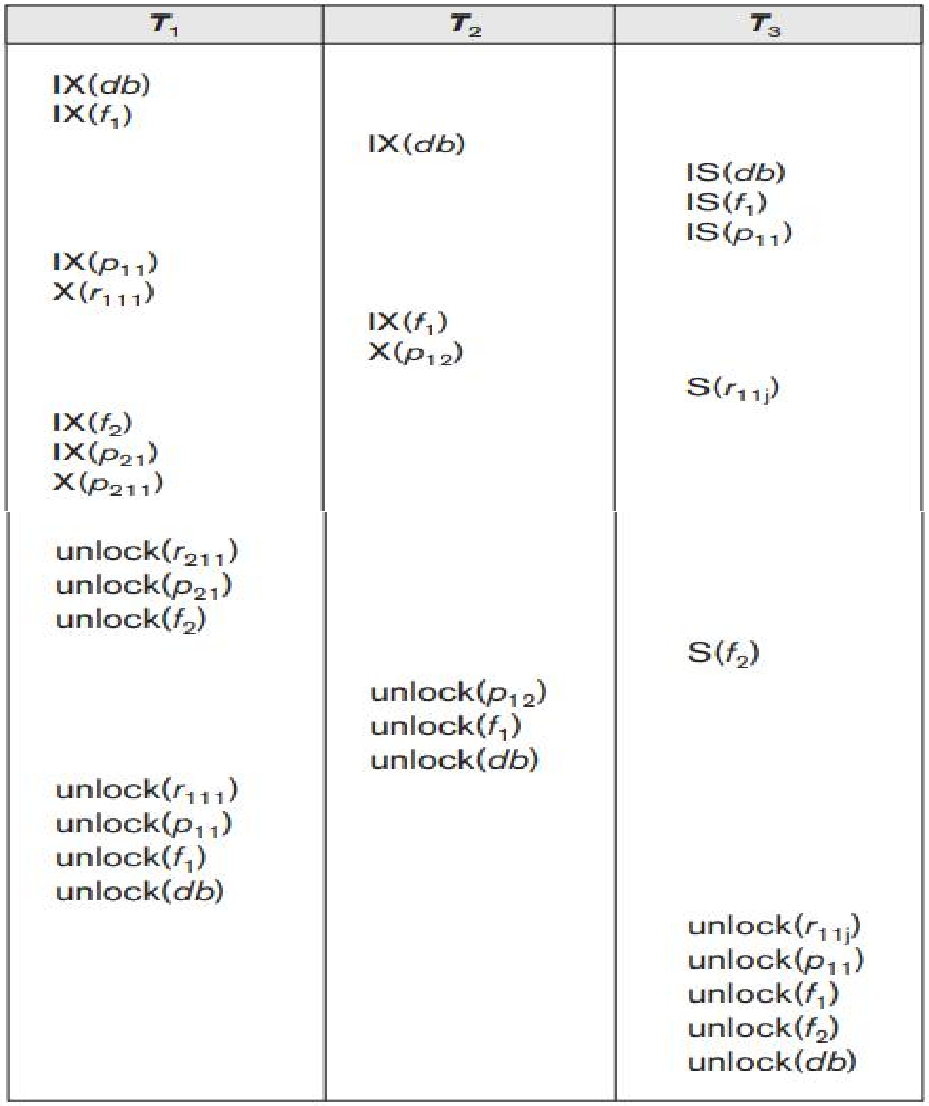
3. **Shared-intention-exclusive (SIX):** It indicates that the current node is locked in shared mode but one or more exclusive locks will be requested on some descendant node(s).

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

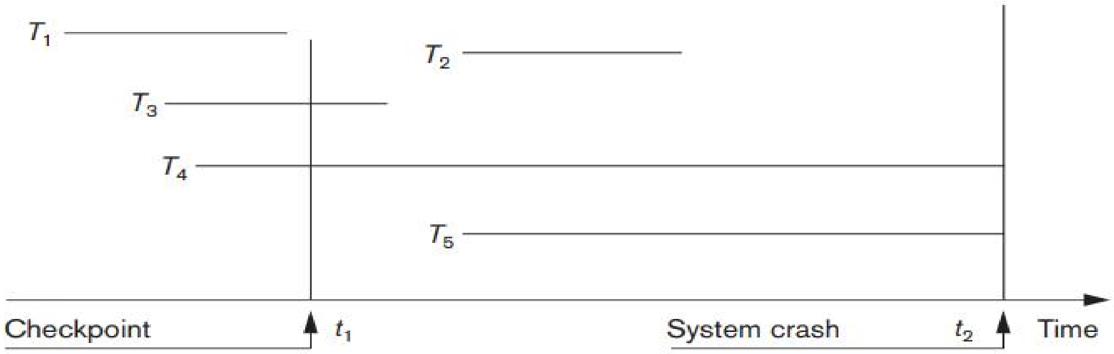
The deferred update thechniques do not physically update the database on disk until after a transaction reaches its commit point, then the updates are recorded in the database. It will not have changed the database in any way, so UNDO is not needed. It may be necesay to REDO the effect of the operation.

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.

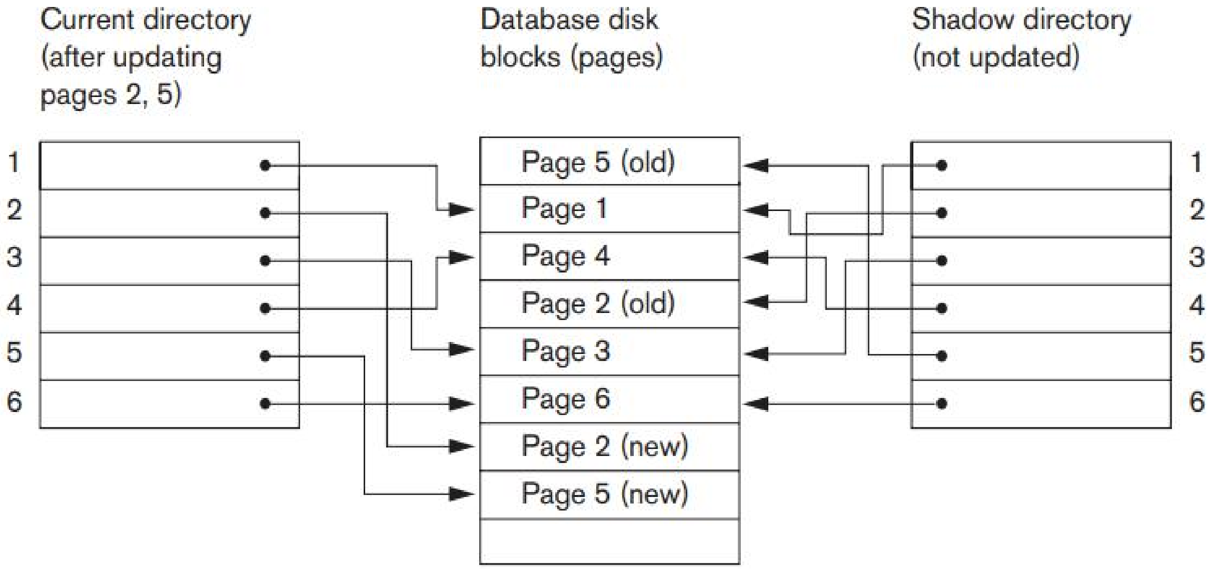
3. Redo all the write_item operations of the committed transactions from the log, in the order in which they were written into the log.

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.

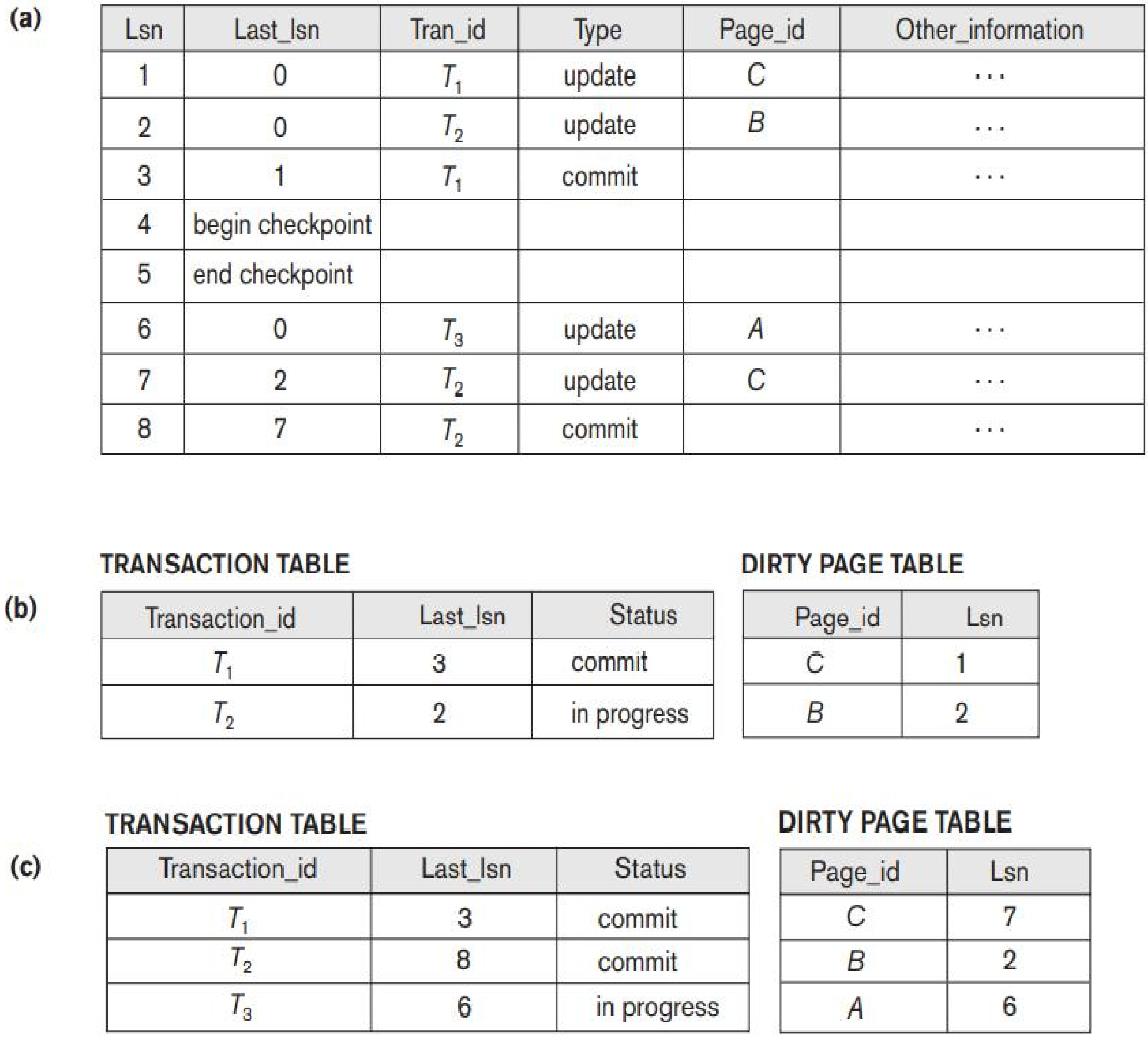
REDO phase: Find the smallest LSN, M of all the dirty pages in the dirty page thable, which indicate the log position where ARIES ready to start REDO phase.

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.

Recovery in Multidatabase System

In some cases, a single transaction, may require access to multiple databases. These databases may even be stored on different types of DBMS, for example, some DBMSs may be relational, Object oriented, hierarchical, or network DBMSs.

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager**, or **Coordinator**, and **Local recovery manager** are participated.

The coordinator usually follows a protocol called the two-phase commit protocol, whose two phases can be stated as follows:

■ **Phase 1:** The coordinator sends a message *prepare for commit* to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a ready to commit or OK signal to the coordinator.

If the local transaction cannot commit for some reason, the participating database sends a cannot commit or not OK signal to the coordinator. If the coordinator does not receive a reply from the database within a certain time out interval, it assumes a not OK response.

■ **Phase 2:** If all participating databases reply OK, and the coordinator's vote is also OK, the transaction is successful, and the coordinator sends a commit signal for the transaction to the participating databases.

Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database.

On the other hand, if one or more of the participating databases or the coordinator have a not OK response, the transaction has failed, and the coordinator sends a message to roll back or UNDO the local effect of the transaction to each participating database. This is done by undoing the local transaction operations, using the log.

Database Backup and Recovery from Catastrophic Failures

The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is a **database backup**, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices.

In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape. Therefore, users do not lose all transactions they have performed since the last database backup.

Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

****