

Database Management System (MDS 505)

Jagdish Bhatta

Unit-1

Fundamental Concept of DBMS

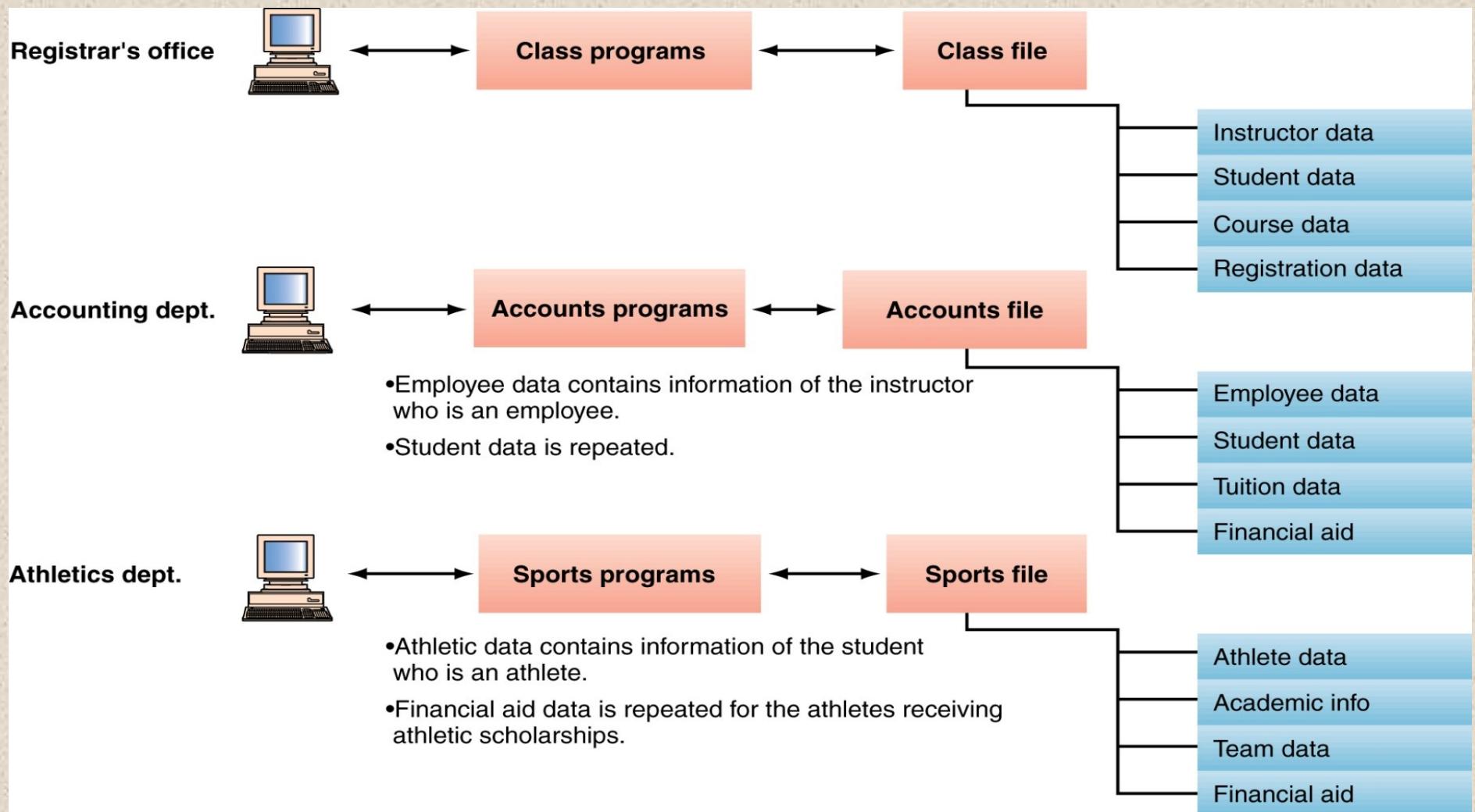
The Data Hierarchy

- **Byte** - 1...8 bits => 1 byte => 1 character
- **Field** - a logical grouping of characters into a word, or a small group of words is called record. It is analogous to column of a table.
- **Record** - a logical grouping of related fields is called record. It is analogous to row of a table.
- **File** - a logical grouping of related records is called file.
- **Database** - a logical grouping of related files is called database.

Traditional File Environment

- ◆ A data file is a collection of logically related records. In the traditional file management environment, each application has a specific data file related to it, containing all the data records needed by the application. It stores data in plain text files and is also called flat file system. This type of database is ideal for simple databases that do not contain a lot of repeated information.
- ◆ Examples include **excel spreadsheet or word data list file**.

Traditional File Environment



Problems in Traditional File Environment

- ◆ Keeping organizational information in a file-processing system has a number of major disadvantages:
 - **Data redundancy:** The address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost.
 - **Data inconsistency:** The various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

Problems in Traditional File Environment

- **Difficulty in accessing data:** Conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner.
- **Data isolation:** Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
- **Integrity problems:** The problem of integrity is the problem of ensuring that the data in database is correct after and before the transaction. For example, the balance of a bank account may never fall below a prescribed amount (say, \$25). When new constraints are added, we have to change the programs to enforce them.

Problems in Traditional File Environment

- **Atomicity problems:** Execution of transactions must be atomic. This means transactions must execute at its entirety or not at all. Consider a program to transfer \$50 from account A to account B . If a system failure occurs during the execution of the program, it is possible that the \$50 was removed from account A but was not credited to account B , resulting in an inconsistent database state. It is difficult to ensure atomicity in a conventional file-processing system.
- **Concurrent-access anomalies:** Concurrent updates may result in inconsistent data. Consider bank account A , containing \$500. If two customers withdraw funds (say \$50 and \$100 respectively) from account A at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state, if the programs executing on behalf of each withdrawal read the old balance

Problems in Traditional File Environment

- **Security problems:** Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult in flat file system.

Database

- ◆ **Database** is a collection of *persistent data*, that have some implicit meaning, and is used by the application systems of some given *enterprise*. A database has the following implicit properties:
 - A database represents some aspects of the real world called universe of discourse.
 - A database is a logically coherent collection of data with some inherent meaning.
 - A database is designed, built and populated with data for a specific purpose.
- ◆ DB may be generated & maintained manually or it may be computerized.

Database

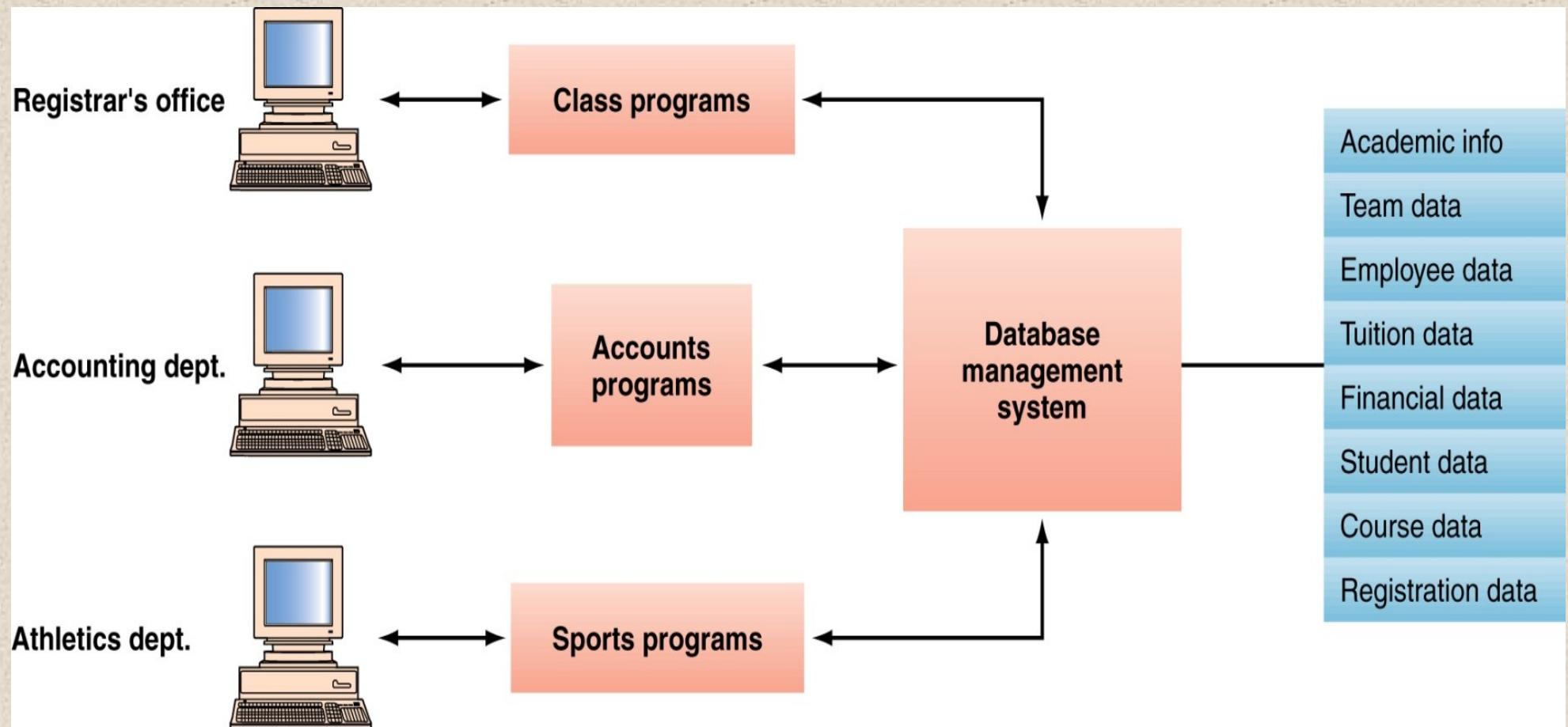
- ◆ A **database** is a collection of related data. By **data**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. Nowadays, this data is typically stored in mobile phones, which have their 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.

Database

E.g.: - *University database* for maintaining information about students, courses and grades in university.

- ◆ Data consists of information about:
 - Students
 - Instructors
 - Classes etc.
- ◆ Application program examples
 - Add new students, instructors, and courses
 - Register students for courses, and generate class rosters
 - Assign grades to students, compute grade point averages (GPA) and generate transcripts

Database



Simplified Database System Environment

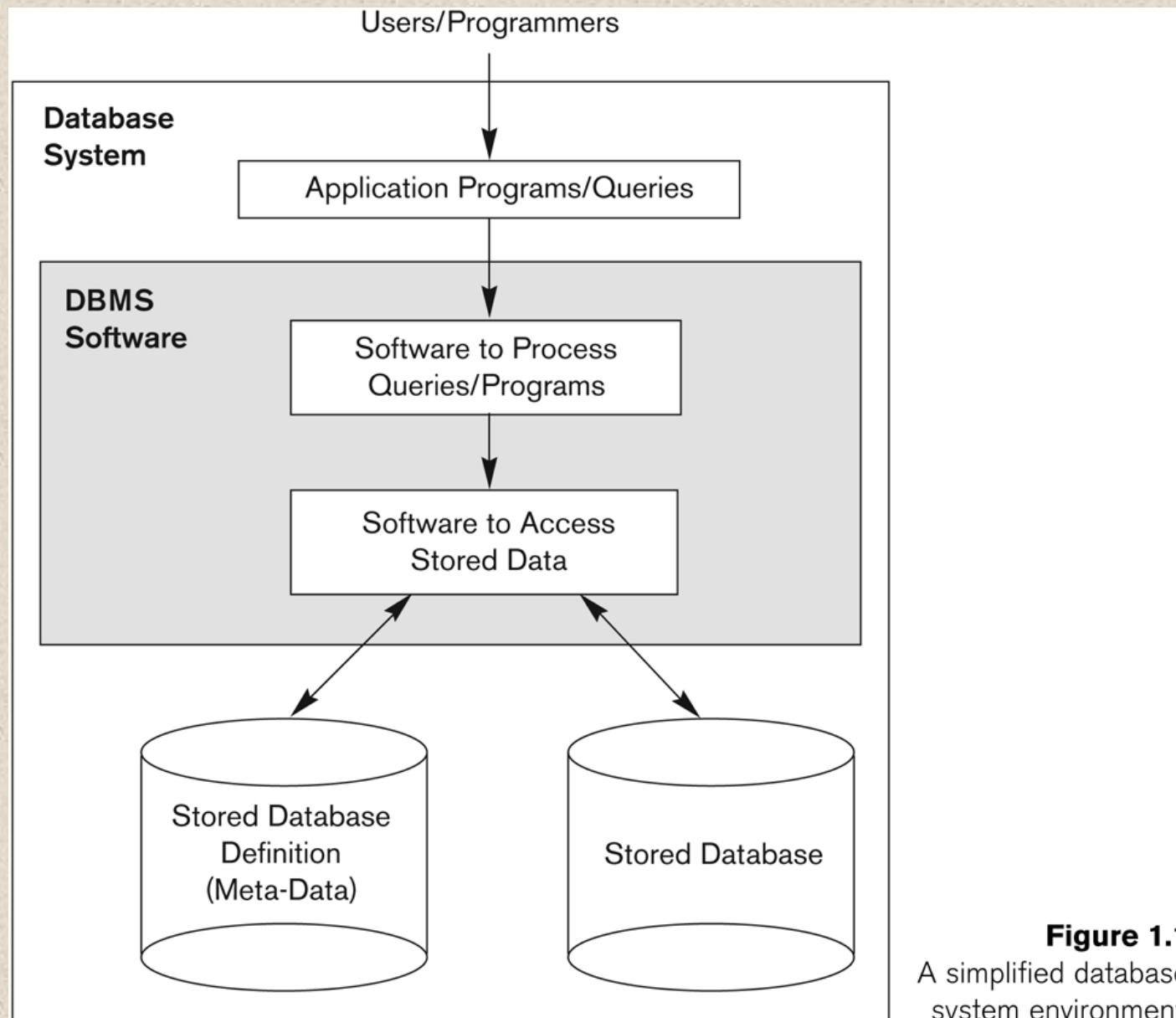


Figure 1.1
A simplified database system environment.

Database Management System [DBMS]

- ◆ The *database management system* (DBMS) is the software that handles all access to the database.
- ◆ It is defined as the collection of interrelated data and a set of programs to access those data. It enables users to create and maintain database. Simply, it is a database manager.
- ◆ This general purpose software system includes process of *defining, constructing, manipulating and sharing* database among various users and applications.

Database Management System [DBMS]

- ◆ **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.

Database Management System [DBMS]

- ◆ 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.
- ◆ 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.

Database Management System [DBMS]

- ◆ DBMS contains information about a particular enterprise
 - Collection of interrelated data
 - Set of programs to access the data
 - An environment that is both *convenient* and *efficient* to use
- ◆ It is not absolutely necessary to use general-purpose DBMS software to implement a computerized database. It is possible to write a customized set of programs to create and maintain the database, in effect creating a *special-purpose* DBMS software for a specific application, such as airlines reservations.
- ◆ For example: MSAccess, Oracle, MySql, MSSql, Cassandra, MongoDB etc.

Example Of A Simple Database

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

Figure 1.2
A database that stores student and course information.

Main Characteristics of the Database Approach

- ◆ **Self-describing nature of a database system:**

- A DBMS **catalog** stores the description of a particular database (e.g. data structures, types, and constraints). The data is stored as **self-describing data** that includes the data item names and data values together in one structure.
- The description is called **meta-data**.
- This allows the DBMS software to work with different database applications.

Example of a simplified database catalog

RELATIONS		
Relation_name	No_of_columns	
STUDENT	4	
COURSE	4	
SECTION	5	
GRADE_REPORT	3	
PREREQUISITE	2	

COLUMNS		
Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
....
....
....
Prerequisite_number	XXXXNNNN	PREREQUISITE

Note: Major_type is defined as an enumerated type with all known majors. XXXXNNNN is used to define a type with four alpha characters followed by four digits

Main Characteristics of the Database Approach

- ◆ **Insulation between programs and data:**

- 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**.
- 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, regardless of how the operations are implemented. This may be termed **program-operation independence**.
- Allows changing data structures and storage organization without having to change the DBMS access programs.

Main Characteristics of the Database Approach (continued)

◆ Data Abstraction:

- 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. Informally, 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, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

Main Characteristics of the Database Approach (continued)

◆ **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.
- For example, one user of the database of Figure 1.2 (in slide number 19) may be interested only in accessing and printing the transcript of each student; the view for this user is shown in Figure 1.5(a) in next slide. A second user, who is interested only in checking that students have taken all the prerequisites of each course for which the student registers, may require the view shown in Figure 1.5(b) in next slide.

Main Characteristics of the Database Approach (continued)

- ◆ Support of multiple views of the data:

TRANSCRIPT					
Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

(a)

COURSE_PREREQUISITES		
Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

(b)

Figure 1.5
Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view.
(b) The COURSE_PREREQUISITES view.

Main Characteristics of the Database Approach (continued)

- ◆ **Sharing of data and multi-user 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)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently.

Advantages of Using the Database Approach

- ◆ **Controlling redundancy** in data storage and in development and maintenance efforts.
 - Sharing of data among multiple users.
- ◆ **Restricting unauthorized access** to data.
- ◆ **Providing persistent storage for program Objects**
 - In Object-oriented DBMSs
- ◆ **Providing Storage Structures** (e.g. indexes) for efficient Query Processing

Advantages of Using the Database Approach

- ◆ Providing **backup and recovery** services.
- ◆ Providing **multiple user interfaces** to different classes of users.
- ◆ Representing **complex relationships** among data.
- ◆ Enforcing **integrity constraints** on the database.
- ◆ Drawing **inferences and actions** from the stored data using rules and triggers

Additional Implications of Using the Database Approach

- ◆ Potential for enforcing standards:
 - This is very crucial for the success of database applications in large organizations. **Standards** refer to data item names, display formats, screens, report structures, meta-data (description of data), Web page layouts, etc.
- ◆ Reduced application development time:
 - Incremental time to add each new application is reduced.

Additional Implications of Using the Database Approach (continued)

- ◆ Flexibility to change data structures:
 - Database structure may evolve as new requirements are defined.
- ◆ Availability of current information:
 - Extremely important for on-line transaction systems such as airline, hotel, car reservations.
- ◆ Economies of scale:
 - Wasteful overlap of resources and personnel can be avoided by consolidating data and applications across departments.

Applications Areas of Database System

- ♦ Databases form an essential part of almost all enterprises. Some applications include:
 - ***Banking:*** For customer information, accounts, and loans, and banking transactions.
 - ***Airlines:*** For reservation and schedule information.
 - ***Universities:*** For student information, course registrations, and grades.
 - ***Credit card transactions:*** For purchase on credit cards and generation of monthly statements.
 - ***Telecommunication:*** For keeping records of call made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

- ***Finance***: For storing information about holdings, sales, and purchase of financial instruments such as stocks and bonds.
- ***Sales***: For customer, product, and purchase information
- ***Manufacturing***: For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.
- ***Human resources***: For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.
- **And many more ...**

Database Management System (MDS 505)

Jagdish Bhatta

Unit-1

Fundamental Concept of DBMS: Data Models

Data Models

- ◆ One fundamental characteristic of the database approach is that it provides some level of data abstraction.
- ◆ **Data abstraction** generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail.
- ◆ A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction. 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.

Data Models

- ◆ Data model is a collection of tools for describing;
 - Data
 - Data relationships
 - Data semantics
 - Data constraints

Data Models

- ◆ **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 storage media, typically magnetic disks.
- ◆ Concepts provided by physical data models are generally meant for computer specialists, not for end users. Between these two extremes is a class of **representational** (or **implementation**) **data models**, which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Data Models

- ◆ 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 in 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. The **entity–relationship model** is a popular high-level conceptual data model.

Data Models

- ◆ Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Representational data models represent data by using record structures and hence are sometimes called **record-based data models**.

Data Models

- ◆ The **object data model** as an example of a new family of higher-level implementation data models that are closer to conceptual data models. A standard for object databases called the ODMG object model has been proposed by the Object Data Management Group (ODMG). Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.
- ◆ **Physical data models** describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An **access path** is a search structure that makes the search for particular database records efficient, such as indexing or hashing. An **index** is an example of an access path that allows direct access to data using an index term or a keyword. It is similar to the index at the end of this text, except that it may be organized in a linear, hierarchical (tree-structured), or some other fashion.

Data Models

- ◆ Another class of data models is known as **self-describing data models**. The data storage in systems based on these models combines the description of the data with the data values themselves. In traditional DBMSs, the description (schema) is separated from the data.
- ◆ These models include **XML** as well as many of the **key-value stores** and **NOSQL systems** that were recently created for managing big data.

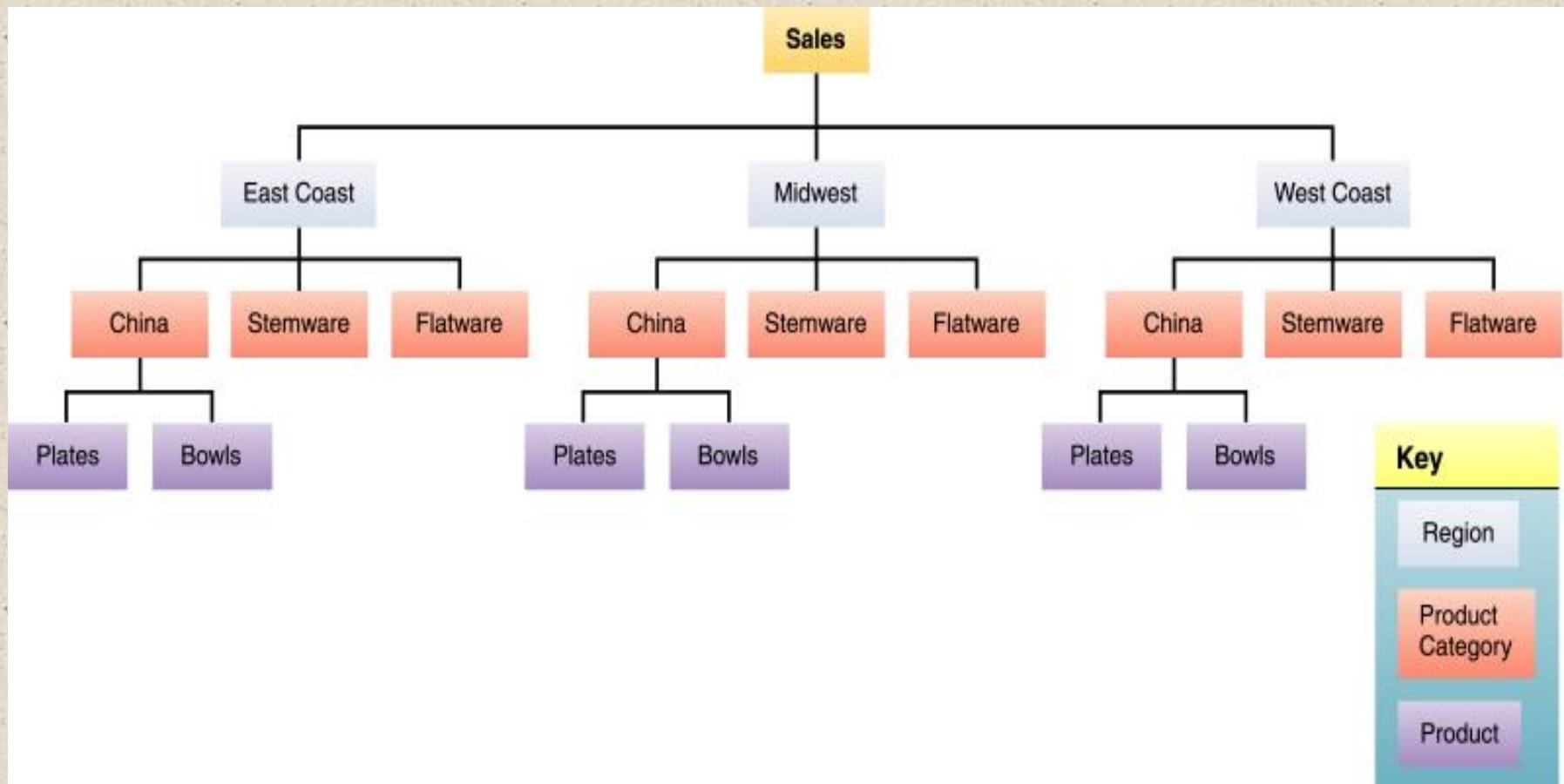
◆ **Hierarchical Data Model:**

- A hierarchical database model is a data model in which the data is organized into a tree-like structure.
- The structure allows representing information using parent/child relationships: each parent can have many children, but each child has only one parent. All attributes of a specific record are listed under an entity type.
- Hierarchical database model rigidly structures data into an inverted “tree” in which each record contains two elements, a single root or master field, often called a key, and a variable number of subordinate fields.

◆ **Hierarchical Data Model:**

- The strongest advantage of the hierarchical database approach is the speed and efficiency with which it can be searched for data.
- The hierarchical model does have problems: Access to data in this model is predefined by the database administrator before the programs that access the data are written. Programmers must follow the hierarchy established by the data structure.
- It can not handle many to many relationship

♦ Hierarchical Data Model:

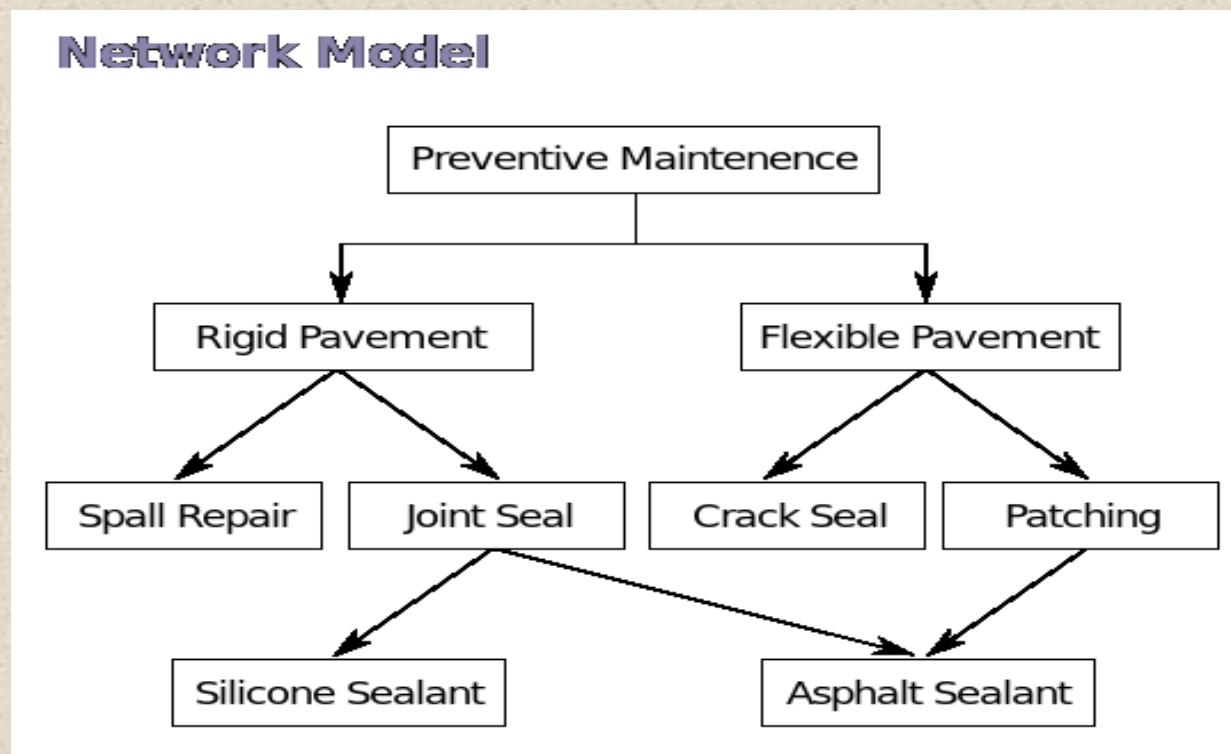


◆ Network Data Model:

- The network model is a database model conceived as a flexible way of representing objects and their relationships.
- Its distinguishing feature is that the schema, viewed as a graph in which object types are nodes and relationship types are arcs.
- Unlike hierarchical model, the network model allows each record to have multiple parent and child records, forming a lattice structure. The network model replaces the hierarchical tree with a graph thus allowing more general connections among the nodes. This model was evolved to specially handle non-hierarchical relationships. Now, same data may exist in two different levels.

◆ Network Data Model:

- Support many-to-many relationship
- Good processing speed but very complex model to design, implement and maintain



◆ Relational Data Model

- Relational data model represents the database as a collection of relations where each relation resembles a table of values with rows and columns.
- A relation may be regarded as a set of **tuples**, also called records. A relation consists of relation schema & relation instance. The **relation schema** specifies relation name & description of tuples (name of attributes, domain). While relation instance corresponds to a table of rows & columns where each row is a tuple & **the column is the attribute**.
- Major advantages of relational model over the older data models are the simple data representation & the ways complex queries can be expressed easily.

◆ Relational Data Model

- Consider a relation representing employee record as;

Employee

Eid	Name	Address	Depart_no
011	Ram Sing	Kathmandu	D01
012	Hari Saha	Pokhara	D02

- The relational model of data permits the database designer to create a consistent, logical representation of information. Consistency is achieved by including declared constraints in the database design, which is usually referred to as the logical schema. The theory includes a process of database normalization whereby a design with certain desirable properties can be selected from a set of logically equivalent alternatives

<i>customer_id</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account_number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer_id</i>	<i>account_number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Fig: A Sample Relational Database

The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model. The relational model is at a lower level of abstraction than the E-R model. Database designs are often carried out in the E-R model, and then translated to the relational model.

◆ Entity-Relationship Model:

- ◆ The *entity-relationship (E-R) model* is a high level data model based on a perception of a real world that consists of collection of basic objects, called *entities*, and of *relationships* among these entities. An *entity* is a thing or object in the real world that is distinguishable from other objects.
- ◆ Entities are described in a database by a set of *attributes*. A *relationship* is an association among several entities. The set of all entities of the same type is called an *entity set* and the set of all relationships of the same type is called a *relationship set*.

The overall logical structure of a database can be expressed graphically by an E-R diagram. The components of this diagram are:

- **Rectangles** (represent entity sets)
- **Ellipses** (represent attributes)
- **Diamonds** (represent relationship sets among entity sets)
- **Lines** (link attributes to entity sets and entity sets to relationship sets)

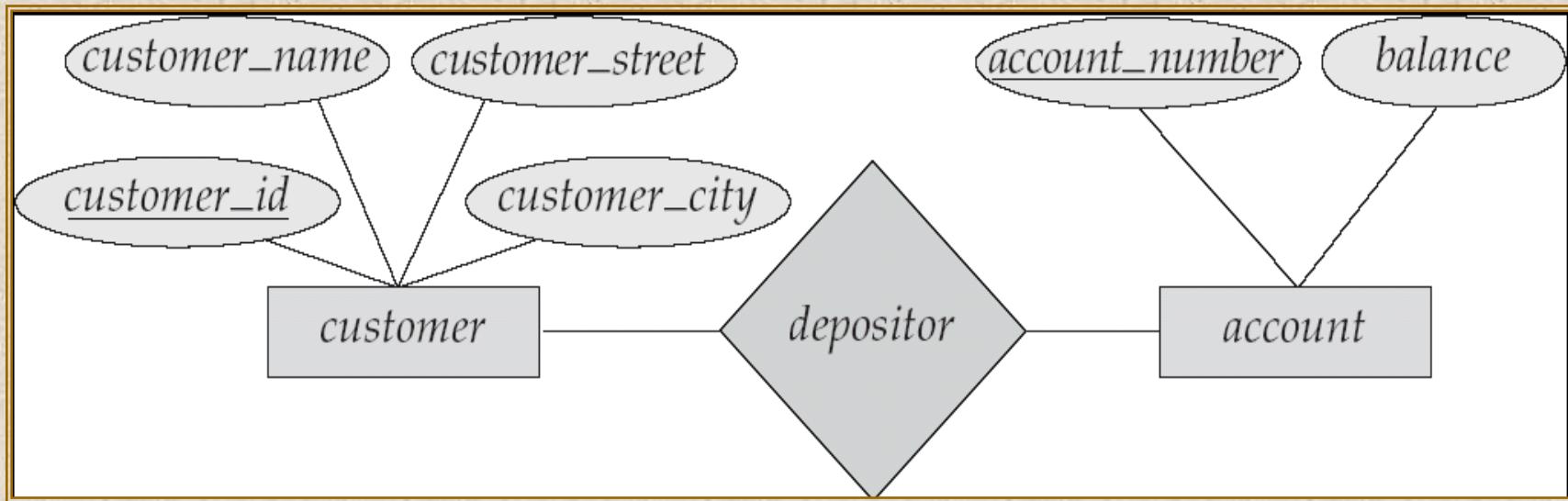


Fig: A Sample E-R Diagram

- ◆ **Object Oriented Database:** An object database (also object-oriented data model) is a database model in which information is represented in the form of objects as used in object-oriented programming.
- ◆ Data model that adds new object storage capabilities to relational databases.

Object-Oriented Model

Object 1: Maintenance Report Object 1 Instance

Date	01-12-01
Activity Code	24
Route No.	I-95
Daily Production	2.5
Equipment Hours	6.0
Labor Hours	6.0

Object 2: Maintenance Activity

Activity Code	
Activity Name	
Production Unit	
Average Daily Production Rate	

Schemas and Instances

- ◆ The description of a database is called the *database schema*, which is specified during database design and is not expected to change frequently.
- ◆ DB Schema
 - captures attributes, relationships, constraints on the data
 - is independent of any application program
- ◆ Databases change over time as data is inserted and deleted. The collection of data stored in the database at a particular moment of time is called an *instance* of the database. Also known as *DB State*.

Schemas and Instances

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
----------------	--------------------	-------

Example Of A Simple Database

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

Figure 1.2
A database that stores student and course information.

Jagdish Bhatta

Schemas and Instances

- ◆ A **schema diagram** displays only *some aspects* of a schema, such as the **names of record types and data items, and some types of constraints**. Other aspects are not specified in the schema diagram; for example, figure 2.1 (in slide number 23) shows neither the data type of each data item nor the relationships among the various files.

Schemas and Instances

- ◆ The actual data in a database may change quite frequently. For example, the database shown in Figure 1.2 (in slide number 24) changes every time we add a new student or enter a new grade. 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. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

Schemas and Instances

- ◆ The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.
- ◆ The DBMS is partly responsible for ensuring that every state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important and the schema must be designed with utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Schemas and Instances

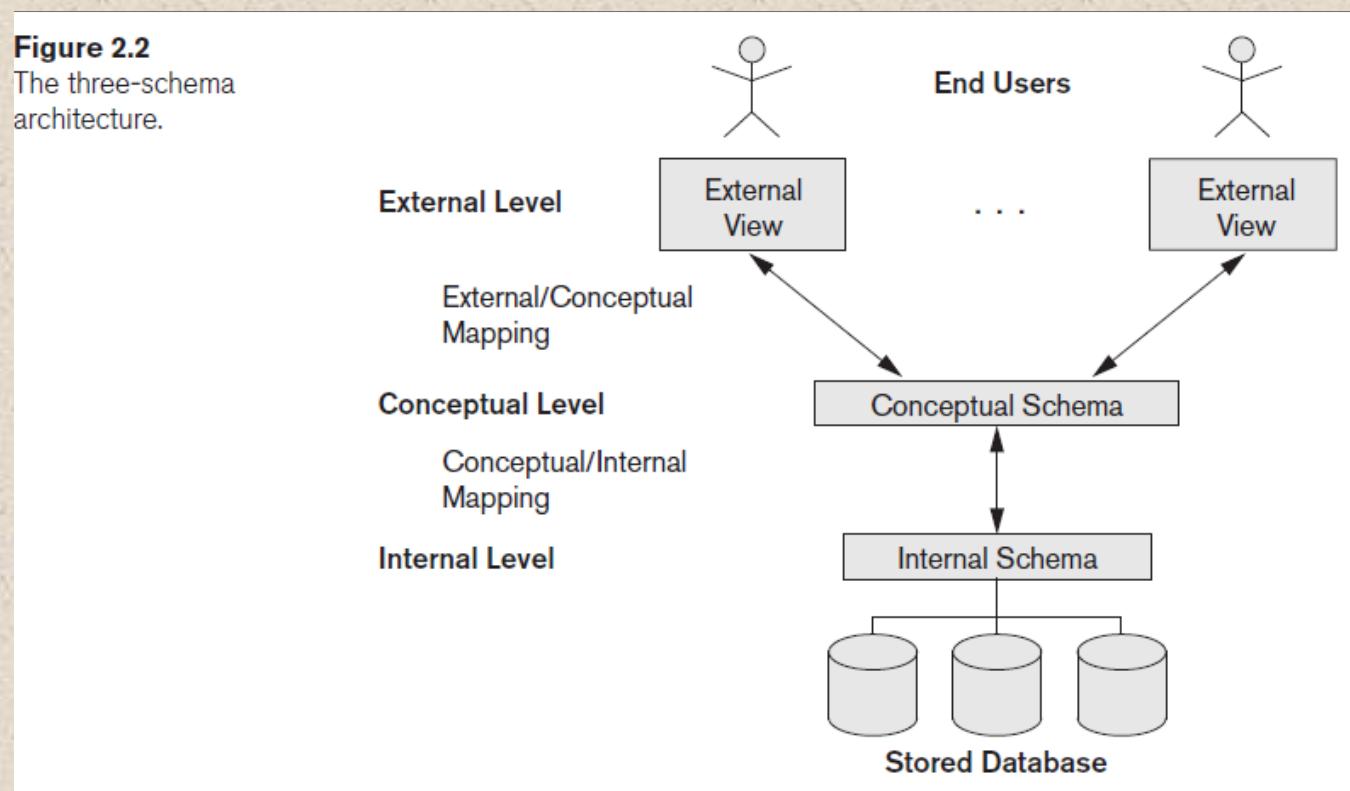
- ◆ Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the Date_of_birth to the STUDENT schema in Figure 2.1 (in slide number 24). This is known as **schema evolution**. Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

Schemas and Instances

- ◆ **Logical Schema** – the overall logical structure of the database
 - Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
 - Analogous to type information of a variable in a program
- ◆ **Physical schema** – the overall physical structure of the database
- ◆ **Instance** – the actual content of the database at a particular point in time
 - Analogous to the value of a variable

Three-Schema Architecture and Data Independence

- It divides the system into three levels of abstraction: the *internal* or *physical level*, the *conceptual level*, and the *external* or *view level*.
- The goal of the three schema architecture is to separate the user applications and the physical database.



Three-Schema Architecture and Data Independence

- ◆ 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.
- ◆ 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. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented. This *implementation conceptual schema* is often based on a *conceptual schema design* in a high-level data model.
- ◆ 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. As in the previous level, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level conceptual data model.

Three-Schema Architecture and Data Independence

- ◆ The DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings**.

Levels of Abstraction

- ◆ **Physical level:** describes how a record (e.g., instructor) is stored. It describes the physical storage structure of the data in the database.
- ◆ **Logical level:** describes data stored in database, and the relationships among the data. It describes the structure of whole database and hides details of physical storage structure. It concentrates on describing entities, data types, relationships, attributes and constraints.

```
type instructor = record
    ID : string;
    name : string;
    dept_name : string;
    salary : integer;
end;
```

- ◆ **View level:** application programs hide details of data types. It includes a number of user views and hence is guided by the end user requirement. It describes only those part of the database in which the users are interested and hides rest of all from those users. Views can also hide information (such as an employee's salary) for security purposes.

Data Independence

The three schema architecture further explains the concept of **data independence**, the capacity to change the schema at one level without having to change the schema at the next higher level.

- **Logical Data Independence**
- **Physical Data Independence**
- ◆ **Logical Data Independence:** The capacity to change the conceptual schema without having to change the external schemas and their associated application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item).

Physical Data Independence: The capacity to change the internal schema without having to change the conceptual schema. Applications depends on logical schema. In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

For example, the internal schema may be changed when certain file structures are reorganized or new indexes are created to improve database performance

Data Independence.....

- ◆ When a schema at a lower level is changed, only the **mappings** between this schema and higher-level schemas need to be changed in a DBMS that fully supports data independence. The higher-level schemas themselves are unchanged.
- ◆ Hence, the application programs need not be changed since they refer to the external schemas.

Database Languages

- ◆ **Database Language:** Language used to interrogate and process data in a relational database
 - **Data definition language (DDL):** Set of statements that describe a database structure (all record types and data set types). DDL statements include creating databases, tables etc. The DDL is used to specify the conceptual schema only.
 - **DCL (Data Control Language):** DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.
 - **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*. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries.

Database Languages

- **Data Manipulation Language (DML):** Instructions used with higher-level programming languages to query the contents of the database, store or update information, and develop database applications. DML statements include accessing data from database.
- There are two main types of DMLs.
 - **High-level or non procedural**
 - **Low-level or procedural**
- A **high-level or nonprocedural** DML can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a precompiler and processed by the DBMS.

Database Languages

- A **low level** or **procedural** DML *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Therefore, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. This type of DML typically retrieves individual records from the database and processes each separately. In this language, the looping, branching etc. Statements are used to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. The programmers use the low-level DML.
- **High-level DMLs, such as SQL**, can specify and retrieve many records in a single DML statement; therefore, they are called **set-at-a-time** or **set-oriented** DMLs. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; therefore, such languages are also called **declarative**.

Database Languages

- Whenever DML commands, 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**.
- On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**

The Database System Environment

- ◆ A database system is partitioned into modules that deal with each of the responsibilities of the overall system.

Database System Environment

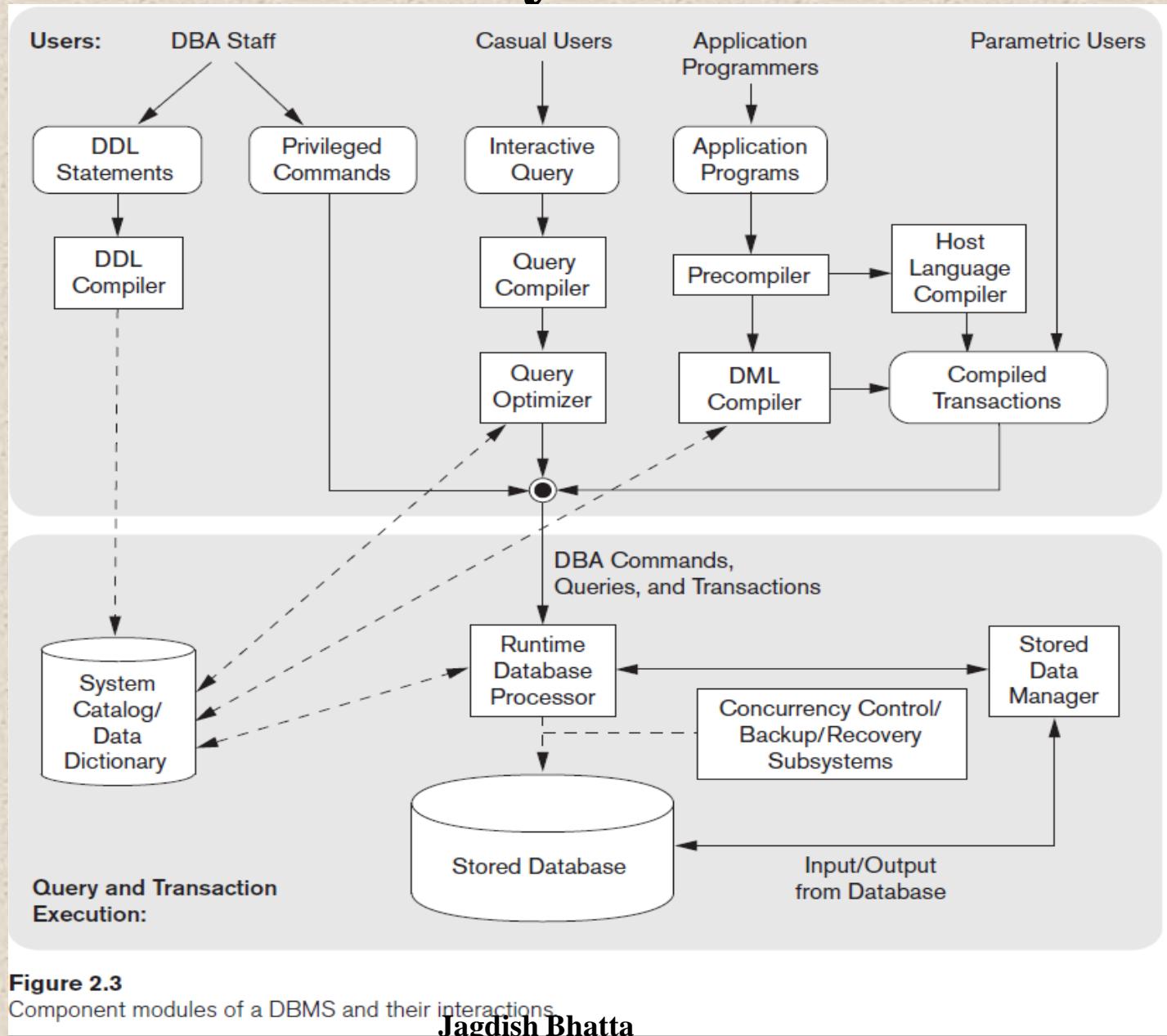


Figure 2.3

Component modules of a DBMS and their interactions

The Database System Environment

- ◆ **DBMS Component Modules:**
- ◆ 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. Many DBMSs have their own **buffer management** module to schedule disk read/write, because management of buffer storage has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.
- ◆ The **DDL compiler** processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints.
- ◆ 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 into an internal form.

The Database System Environment

- ◆ **DBMS Component Modules:**
- ◆ This internal query is subjected to query optimization. Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of efficient search algorithms during execution.
- ◆ 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. The rest of the program is sent to the host **language compiler**.
- ◆ The **runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. 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. **Concurrency control** and **backup and recovery systems** are integrated into the working of the runtime database processor for purposes of transaction management.

The Database System Environment

- ◆ **Database System Utilities:** Most DBMSs have **database utilities** that help the DBA manage the database system.
 - **Loading.** A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database.
 - **Backup.** A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure.
 - **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations and create new access paths 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.

The Database System Environment

- ◆ **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. In addition to storing catalog information about schemas and constraints, 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 and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.
- ◆ 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

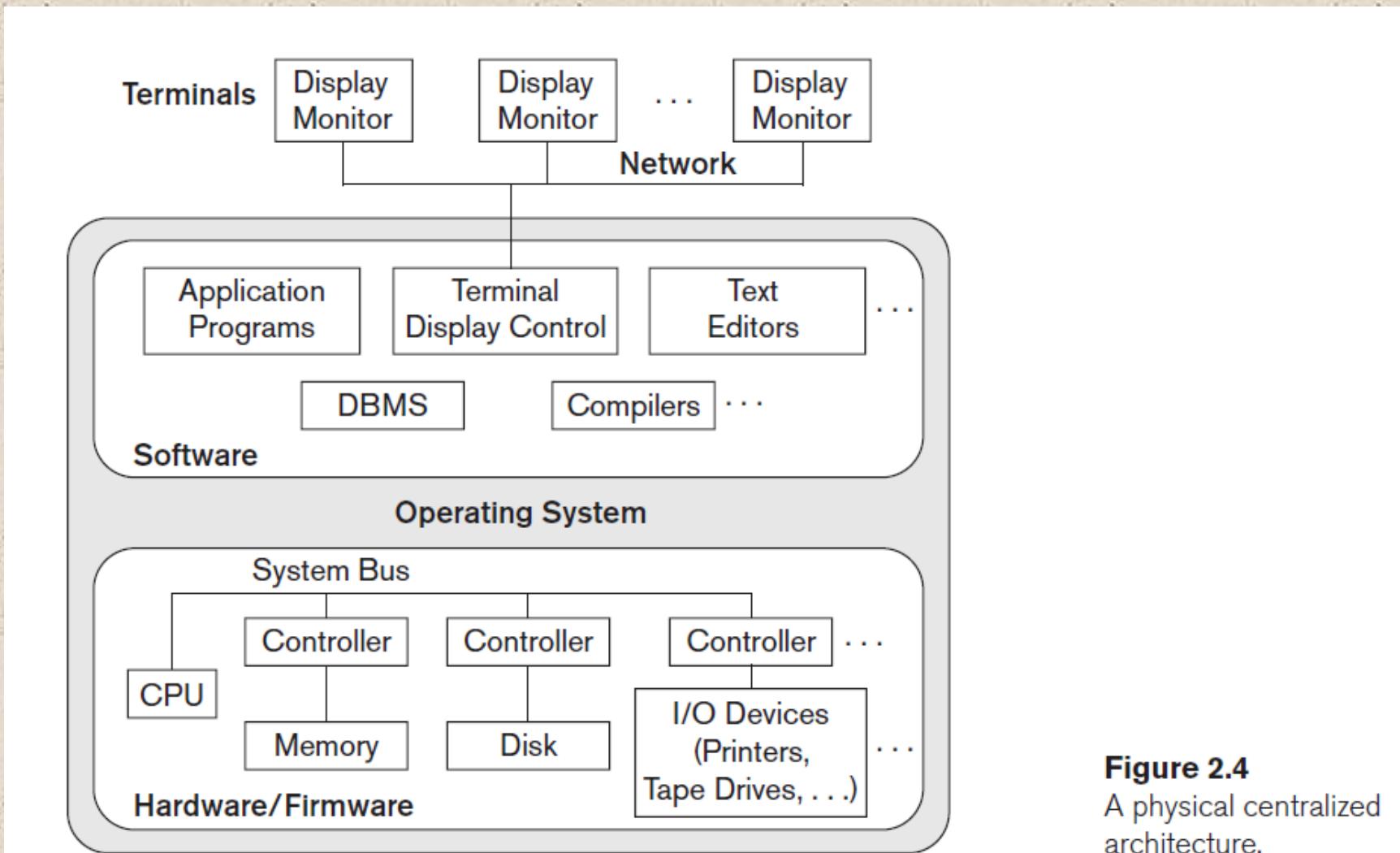
Centralized and Client/Server Architectures for DBMSs

- ◆ Database systems structure:
 - Centralized databases
 - One to a few cores, shared memory
 - Client-server,
 - One server machine executes work on behalf of multiple client machines.

Centralized and Client/Server Architectures for DBMSs

- ◆ **Centralized Database:** 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.

Centralized and Client/Server Architectures for DBMSs



Centralized and Client/Server Architectures for DBMSs

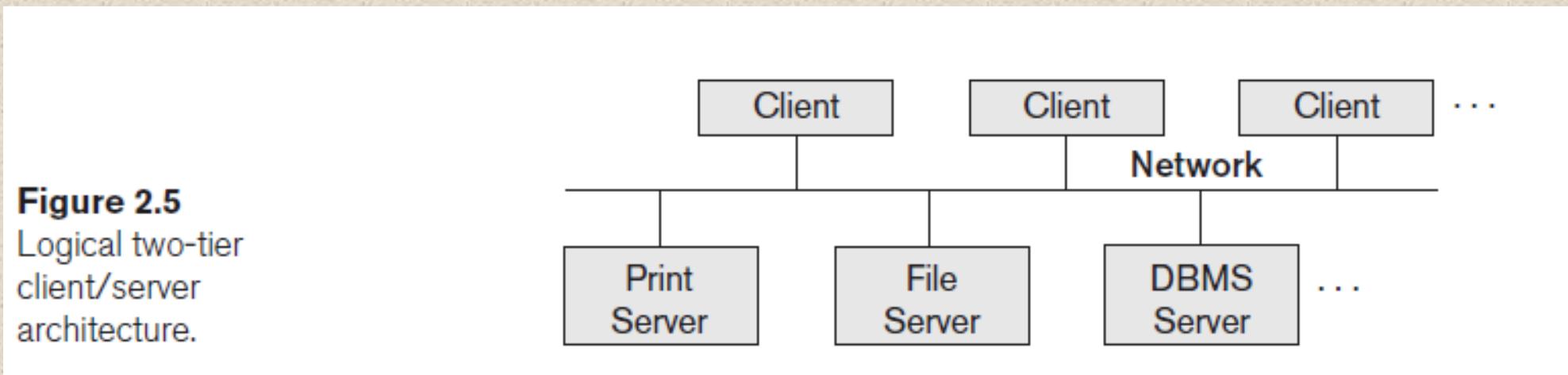
- ◆ **Basic Client Server Architecture:**
- ◆ The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network.
- ◆ The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine.
- ◆ **Web servers** or **e-mail servers** also fall into the specialized server category.

Centralized and Client/Server Architectures for DBMSs

- ◆ **Basic Client Server Architecture:**
- ◆ The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications.
- ◆ A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at the client, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access.

Centralized and Client/Server Architectures for DBMSs

◆ Basic Client Server Architecture:



Centralized and Client/Server Architectures for DBMSs

- ◆ Underlying client/server framework database applications are usually partitioned into two, three or n-parts.
 - **Two-tier architecture** -- the application resides at the client machine, where it invokes database system functionality at the server machine
 - The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS.

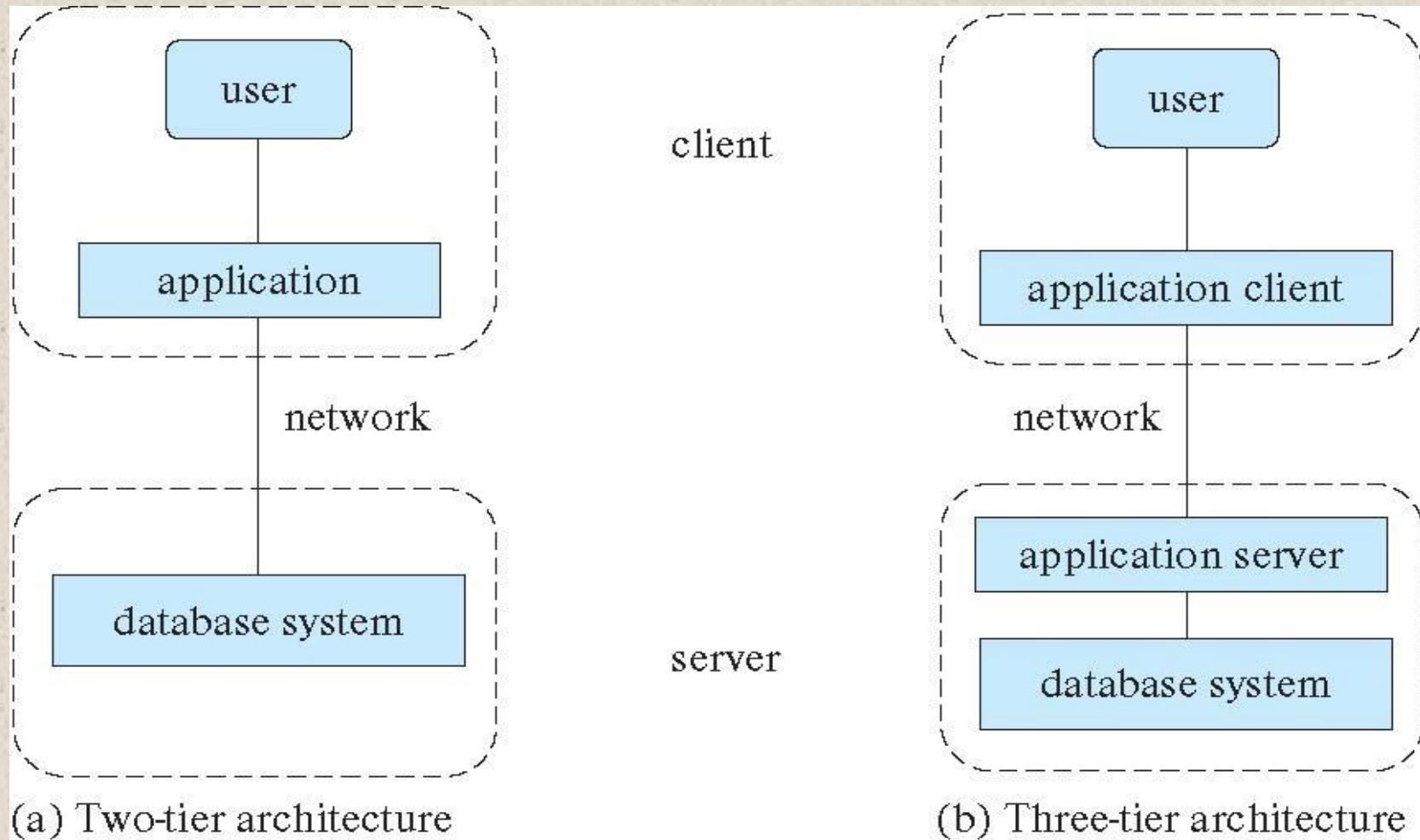
Centralized and Client/Server Architectures for DBMSs

- **Three-tier architecture** -- the **client machine acts as a front end** and does not contain any direct database calls. The client end communicates with an **application server**, usually through a forms interface. The application server in turn communicates with a **database system** to access data.
- 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 (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain **user 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.

Client/Server Database Architecture

- **n-tier architecture** -- the client machine acts as a front end and does not contain any direct database calls.
 - The client end communicates with an application server, usually through a forms interface.
 - The application server in turn communicates with n-tiers/layers of servers

Two-tier and three-tier architectures



Relational Algebra

- ◆ **The relational algebra** is a procedural query language. A relational algebra query describes a procedure for computing the output relation from the input relations by applying the relational algebra operators.
- ◆ The relational algebra defines a set of operators from set theory as *Union*, *Intersection*, *Set Difference*, *Cartesian Product*. Similarly, other operations as *Select*, *Project* & *Join* are incorporated with in the algebra operations. These operations can be categorized as unary or binary according as the number of relation on which they operate on.
- ◆ **Importance of Relational Algebra:**

The relational algebra is very important for several reasons:

- *First*, it provides a formal foundation for relational model operations.
- *Second* and perhaps more important, it is used as a basis for implementing and optimizing queries in relational DBMS (RDBMS).
- *Third*, some of its concepts are incorporated into the SQL standard query language for RDBMS

Codd's Rule

- ◆ Dr Edgar F. Codd, after his extensive research on the Relational Model of database systems, came up with twelve rules of his own, which according to him, a database must obey in order to be regarded as a true relational database.
- ◆ Rule 1: Information Rule
 - The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.
- ◆ Rule 2: Guaranteed Access Rule
 - Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.
- ◆ Rule 3: Systematic Treatment of NULL Values
 - The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Codd's Rule

◆ Rule 4: Active Online Catalog

- The structure description of the entire database must be stored in an online catalog, known as **data dictionary**, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

◆ Rule 5: Comprehensive Data Sub-Language Rule

- A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

◆ Rule 6: View Updating Rule

- All the views of a database, which can theoretically be updated, must also be updatable by the system

Codd's Rule

- ◆ Rule 7: High-Level Insert, Update, and Delete Rule
 - A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.
- ◆ Rule 8: Physical Data Independence
 - The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.
- ◆ Rule 9: Logical Data Independence
 - The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application.

Codd's Rule

- ◆ Rule 10: Integrity Independence
 - A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.
- ◆ Rule 11: Distribution Independence
 - The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.
- ◆ Rule 12: Non-Subversion Rule
 - If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Database Management System (MDS 505)

Jagdish Bhatta

Unit-1

Fundamental Concept of DBMS:ER Model

High-Level Conceptual Data Models for Database Design

- ◆ **Precise Database:**
- ◆ Need to come up with a methodology to ensure that each of the relations in the database is “good”, “complete” and “precise”.
- ◆ Two ways of doing so:
 - Entity Relationship Model
 - Models an enterprise as a collection of *entities* and *relationships*
 - Represented diagrammatically by an *entity-relationship diagram*:
 - Normalization Theory
 - Formalize what designs are bad, and test for them

High-Level Conceptual Data Models for Database Design

- ◆ **Database Design Considerations:**
 - Data Constraints and Relational Database Design
 - Usage Requirements: Queries, Performance
 - Throughput, Response Time
 - Authorization Requirements
 - Data Flow

Database Design Considerations

- ◆ **Be aware about avoiding:**
 - Redundancy
 - Incompleteness

Entities Relationship Model

- ◆ The *entity-relationship (E-R) model* is a high level data model based on a perception of a real world that consists of collection of basic objects, called *entities*, and of *relationships* among these entities. An *entity* is a thing or object in the real world that is distinguishable from other objects.

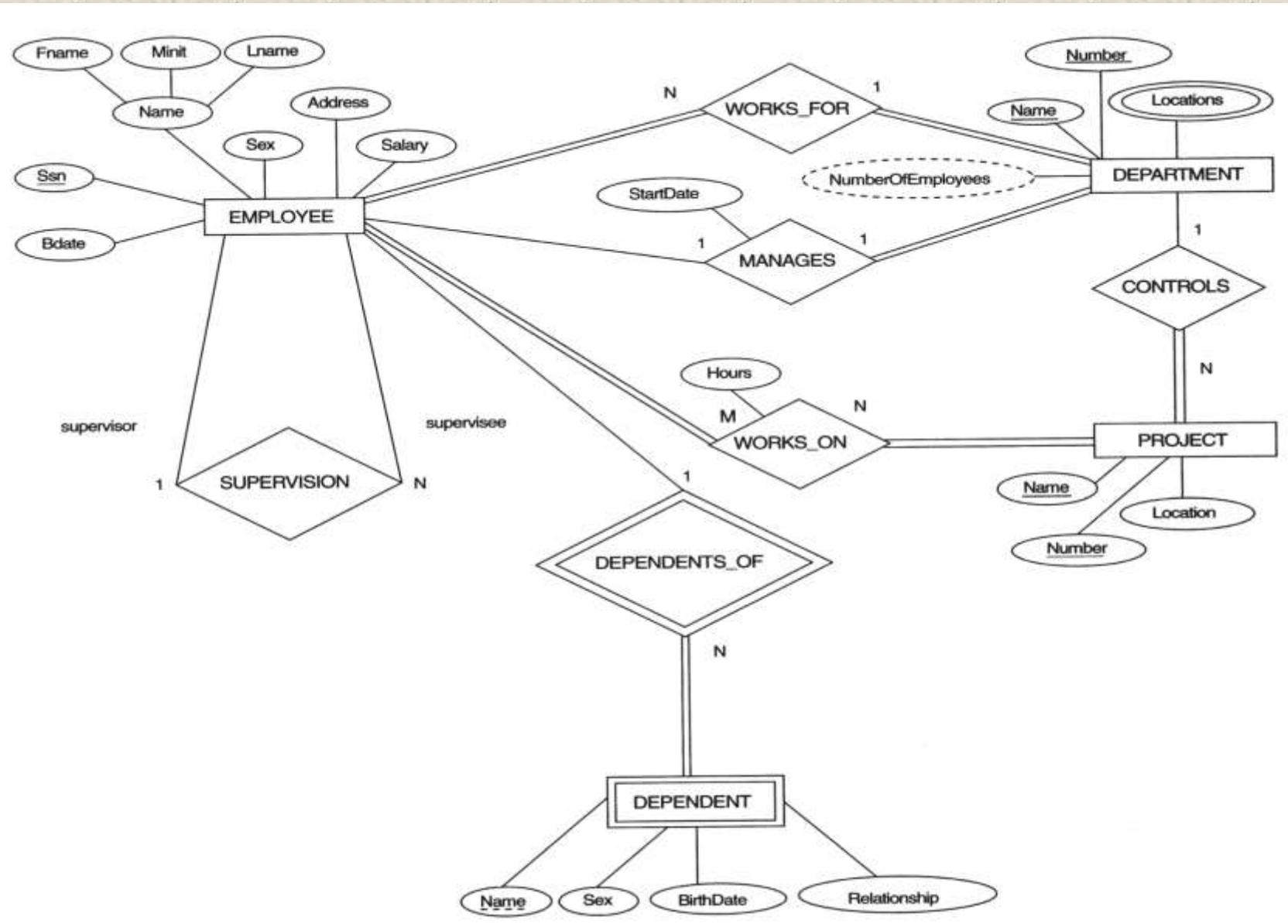
Entities are described in a database by a set of *attributes*. A *relationship* is an association among several entities. The set of all entities of the same type is called an *entity set* and the set of all relationships of the same type is called a *relationship set*.

- ◆ *ER Model is logical modeling of a database.*

Entities and Attributes

- ◆ **Entity** is a thing in real world with and independent existence that is distinguishable from all other objects. An Entity may be an object with physical existence like person, vehicle etc. or may be an object with conceptual existence like account, course etc.
- ◆ Each entities have certain kind of properties that gives proper identification to each of entity. Thus, **attributes** are the properties that describe the entity. Each entity has value for each of its attributes. Thus, for each attribute there is a set of permitted values known as **Domain**. Hence attribute of an entity set can be defined as a function that maps from the entity set to a domain. Example:
 - Entity:- Employee
 - Attributes:- Employee_id, Name, Salary, Age, Phone
 - An instance of entity Employee will be defined once certain values will be assigned to the given attributes.

ER Diagram: An Example



Attribute Types.....

- ◆ Attributes can be of following types;
 - Simple vs. Composite
 - Single Valued vs. Multi-valued
 - Stored vs. Derived
 - Null Valued
 - Complex
- ◆ Simple Vs. Composite:-
 - Simple are those that are not divisible. They are also known as “atomic attributes”.
 - Composite attributes are those that are made up of one or more simple or composite attributes & hence are divisible to smaller subparts. Composite attributes can form a hierarchy.
 - Eg: Address can be the composite attribute as it can be subdivided into Streetaddress, City, State.

Similarly, Name can be divided as Fname, Lname. But Fname, Lname themselves are simple.

Attribute Types.....

- ◆ **Single Valued vs. Multi-valued:-**
 - Single valued are those having just a single value, for a particular entity, from the permitted domain.
 - Multi-valued are those that can have a set of values for same entity.
 - Eg: Roll_no, age can be considered as single valued. While Phone_no, College_degree can have multiple values, so are multivalued.
- ◆ **Stored vs. Derived:-**
 - An attribute whose value can be derived from or inferred from the value of another attribute, then it is derived otherwise it is stored. For example, Age attribute is derived as it can be inferred from the Date_of_birth attribute which is stored.
- ◆ **Null Valued:-**
 - Are those attributes that do not have any applicable values. The null values can also be used if we do not know the value of an attribute for a particular entity.

Attribute Types.....

- ◆ **Complex:-**
 - 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**. For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person. Both phone and address themselves are composite attributes.

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

Entities types and Entity sets

- ◆ **Entity Types:** Entity type defines a collection of entities that have same attributes, but each entity has its own value for each attributes. An entity type describes the schema for a set of entities that share same structure. Each entity type in the database is described by its name and attributes. An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure.
- ◆ **Entity Sets:** An entity set is a set of entities of the same type that share the same properties, or attributes i.e. the collection of all the entities of particular type in the database at any instance is called the **entity set or entity collection**. The entity set is usually referred to using the same name as entity type.
- ◆ In the process of modeling, we often use the term *entity set in the abstract*, without referring to a particular set of individual entities. We use the term **extension** of the entity set to refer to the actual collection of entities belonging to the entity set. The actual instances of employee form extension of the entity employee.

Entities types and Entity sets

◆ Example:

Entity Type Name:	EMPLOYEE	COMPANY	Figure 3.6
Entity Set: (Extension)	Name, Age, Salary e_1 • (John Smith, 55, 80k) e_2 • (Fred Brown, 40, 30K) e_3 • (Judy Clark, 25, 20K) ⋮	Name, Headquarters, President c_1 • (Sunco Oil, Houston, John Smith) c_2 • (Fast Computer, Dallas, Bob King) ⋮	Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

e4

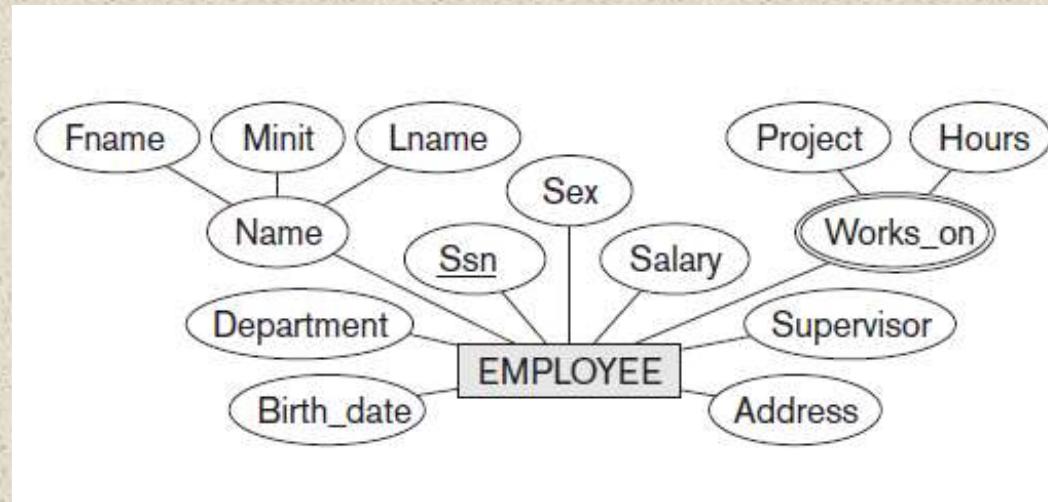
- ◆ **(John Smith, 55, 80k}**

Key Attributes of Entity Types

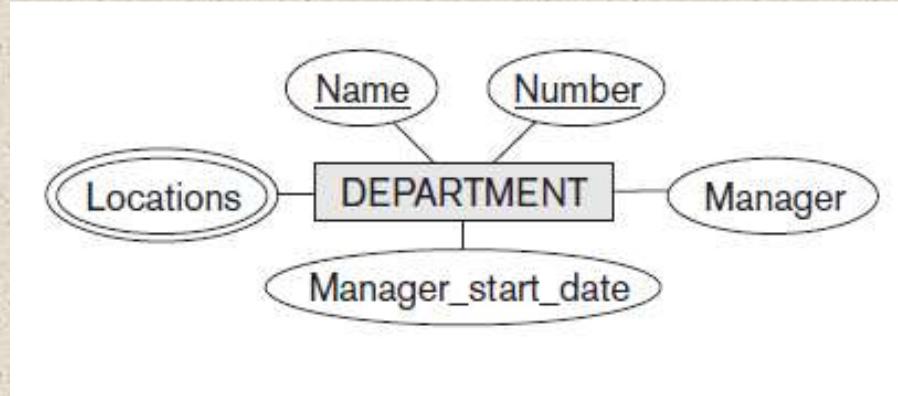
- ◆ An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely.
- ◆ Sometimes several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a *composite attribute* and designate it as a key attribute of the entity type. Notice that such a composite key must be *minimal*; that is, all component attributes must be included in the composite attribute to have the uniqueness property.

Keys Attributes of Entity Types

- ◆ In the entity employee, Ssn is the primary key attribute.

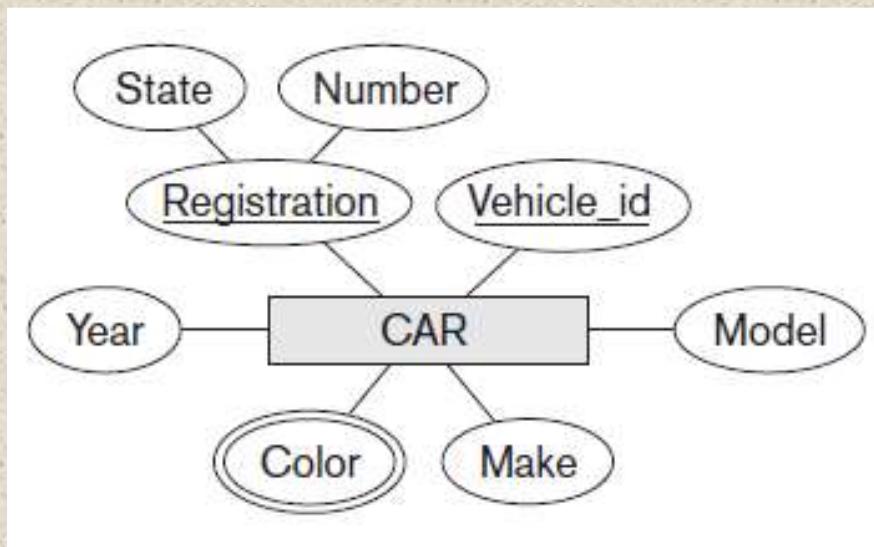


- ◆ In the entity Department, both Name and Number are the primary key attributes.

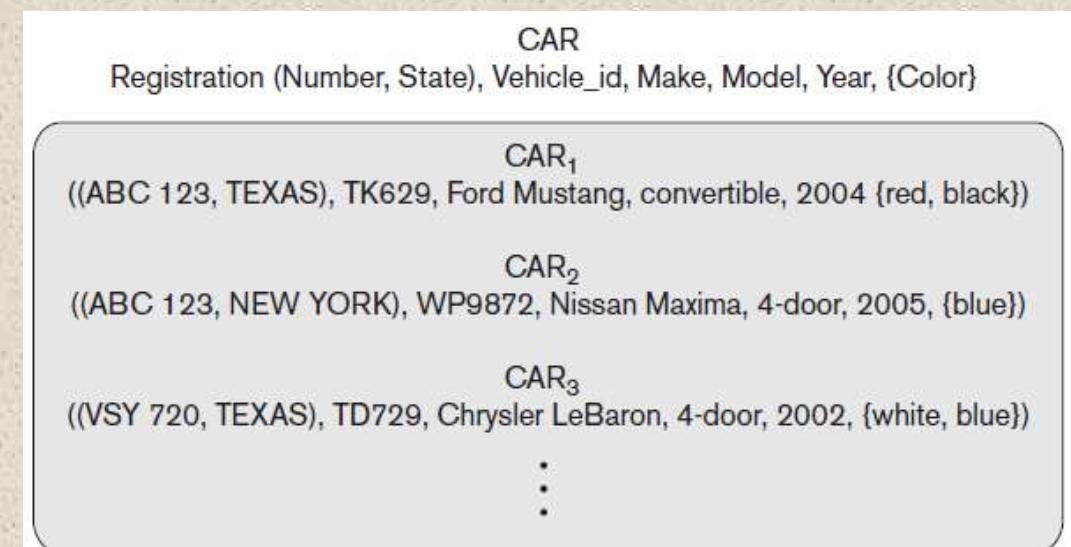


Keys Attributes of Entity Types

- In the entity Car, registration and vehicle_id are the primary key attributes. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own.



The CAR entity type



CAR entity set with three entities.

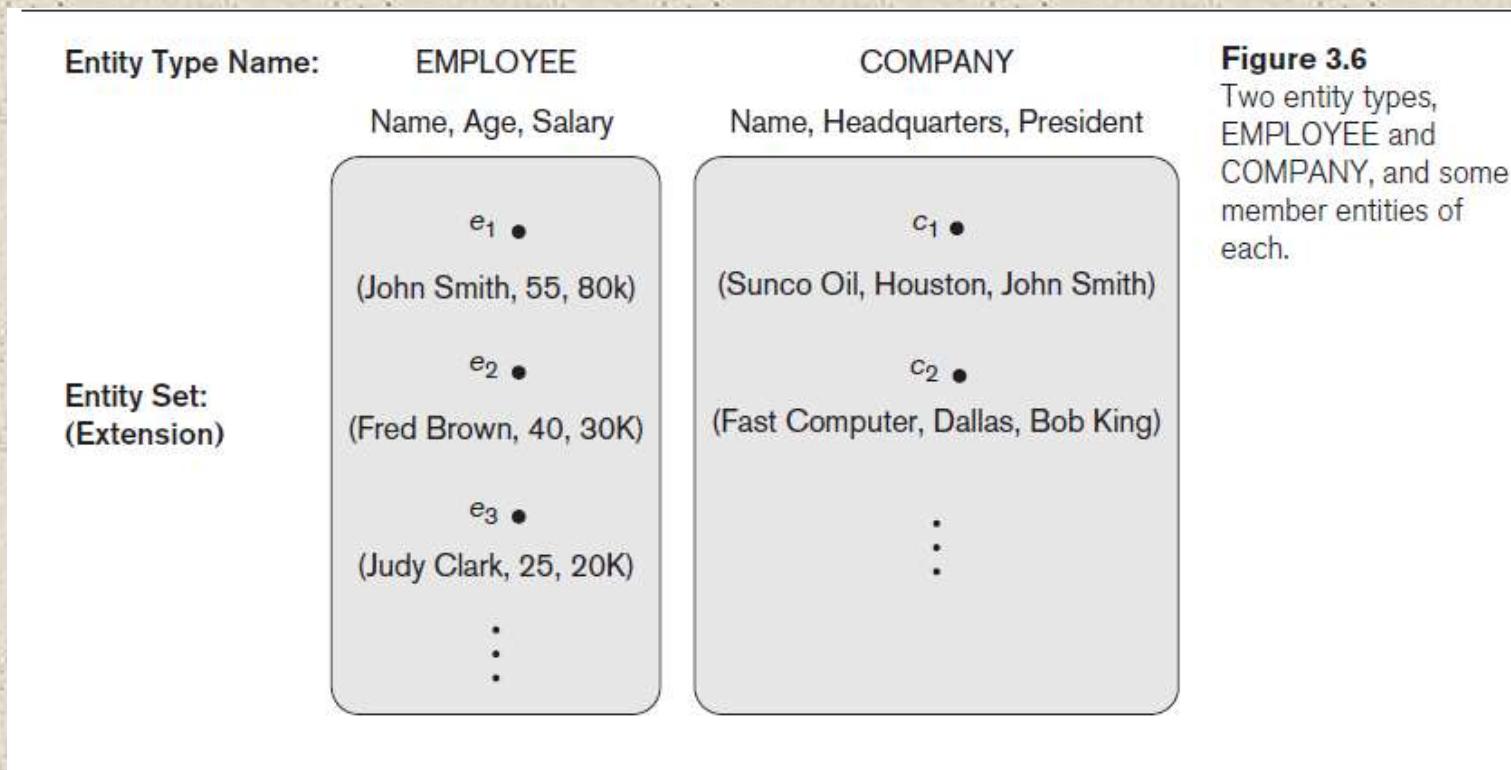
Value Sets (Domains) of Attributes

- ◆ **Value Sets (Domains) of Attributes.** Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity. Value sets are not typically displayed in basic ER diagrams and are similar to the basic **data types** available in most programming languages, such as integer, string, Boolean, float, and so on.
- ◆ Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set $P(V)$ of V :

$$A : E \rightarrow P(V)$$

Value Sets (Domains) of Attributes

- In Figure 3.6, 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.



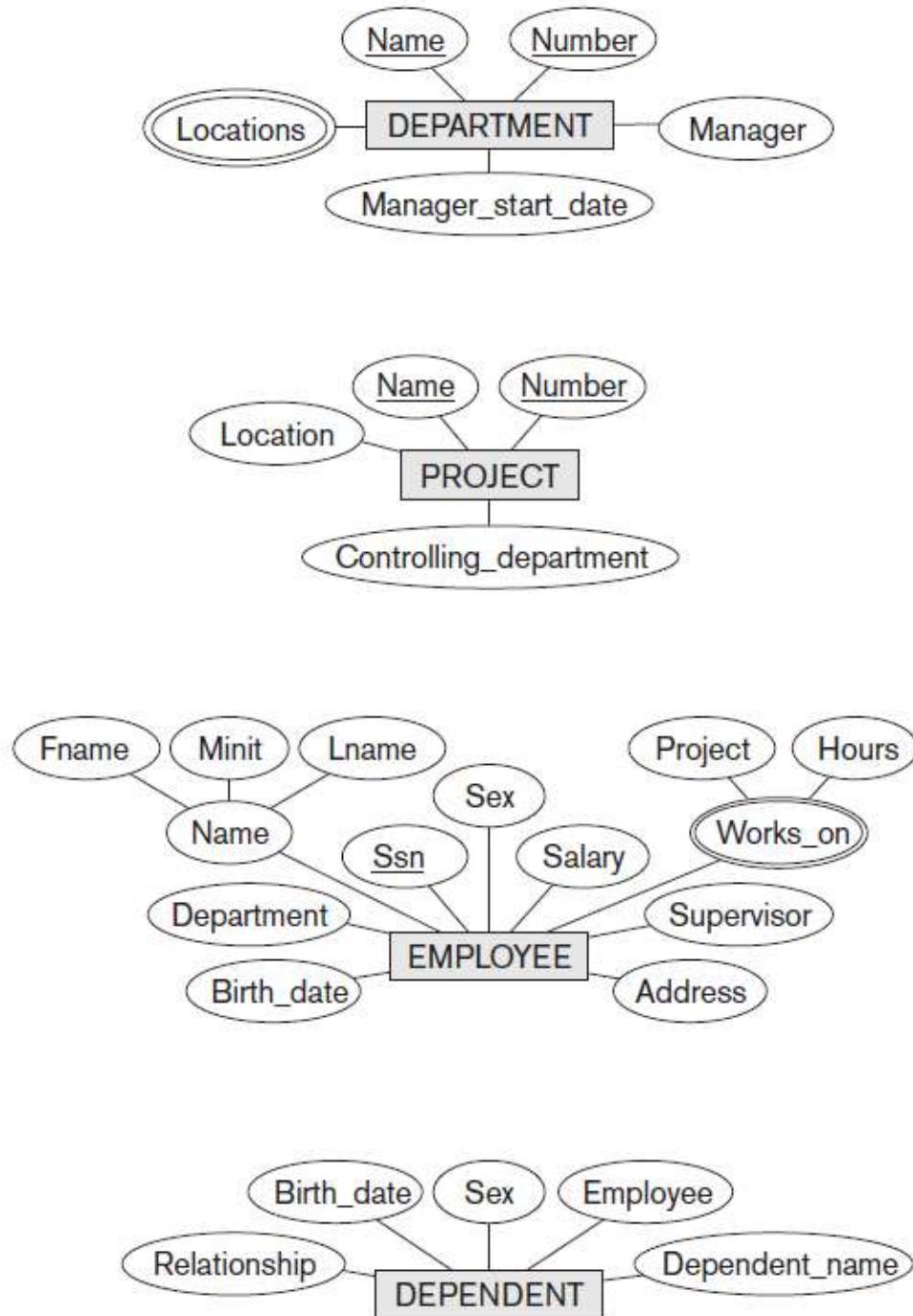
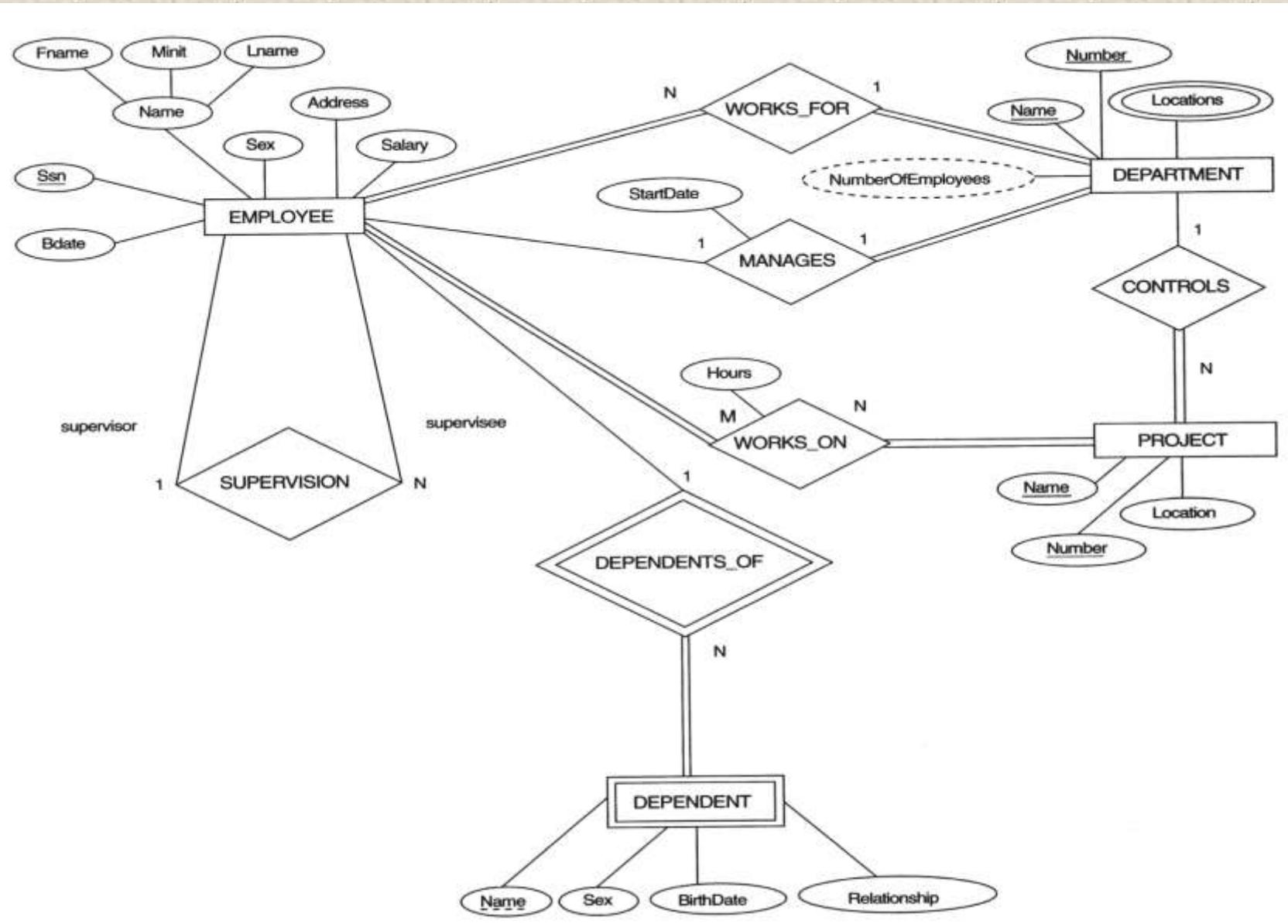


Figure 3.8
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

ER Diagram: An Example



Relationship Types, Relationship Sets, Roles

- ◆ **Relationship** is an association among several entities. **Relationships** can be thought of as *verbs*, linking two or more nouns. Examples: a teaches relationship between a teacher and a student, a *supervises* relationship between an employee and a department etc.

Relationship Types, Relationship Sets, Roles

- ◆ A **relationship type** R among n entity types E_1, E_2, \dots, E_n defines a set of associations—or a **relationship set**—among entities from these entity types. Similar to 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** r_i , where each r_i associates n individual entities (e_1, e_2, \dots, e_n) , and each entity e_j in r_i is a member of entity set E_j , $1 \leq j \leq n$. Hence, a relationship set is a mathematical relation on E_1, E_2, \dots, E_n ; alternatively, it can be defined as a subset of the Cartesian product of the entity sets $E_1 \times E_2 \times \dots \times E_n$. Each of the entity types E_1, E_2, \dots, E_n is said to **participate** in the relationship type R ; similarly, each of the individual entities e_1, e_2, \dots, e_n is said to **participate** in the relationship instance $r_i = (e_1, e_2, \dots, e_n)$.

Relationship Types, Relationship Sets, Roles

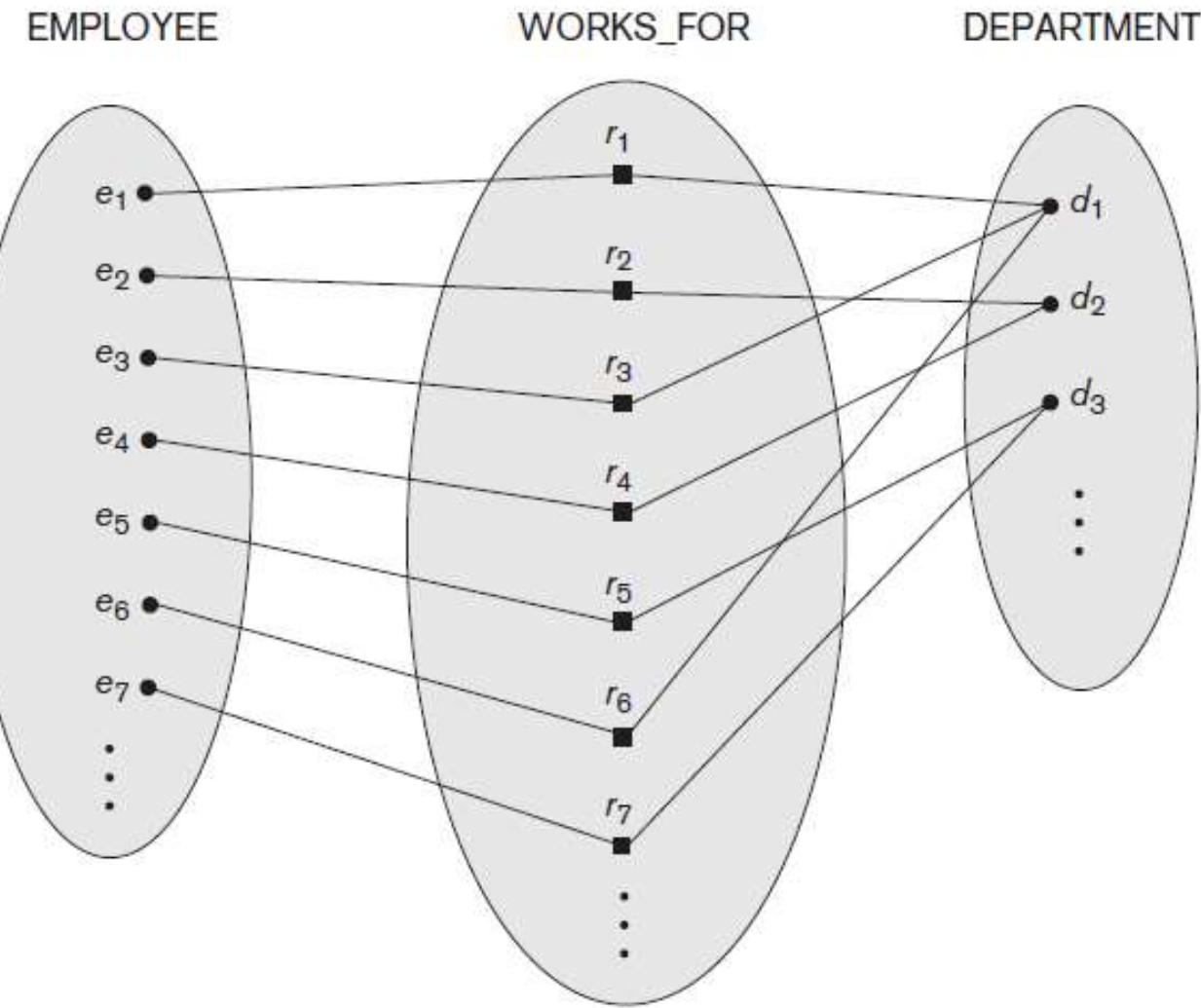


Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

Relationship Types, Relationship Sets, Roles

- ◆ **Degree of relationship type:-** The **degree of relationship type** is the number of participating entity types. Relationships can generally be of any degree, but the most common ones are binary relationships. i.e. having two participating entity types.
 - having three participating entity types - Ternary relationship
 - having n participating entity types – n-ary relationship
- ◆ The function that an entity plays in a relationship is called its **role**. **Roles** are normally explicit and not specified. They are useful when the meaning of a relationship set needs clarification. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and it helps to explain what the relationship means.
- ◆ When some entity type participates more than once in a relationship type in different roles, then we term it as a **recursive relationship or Self-Referencing**.

Relationship Types, Relationship Sets, Roles

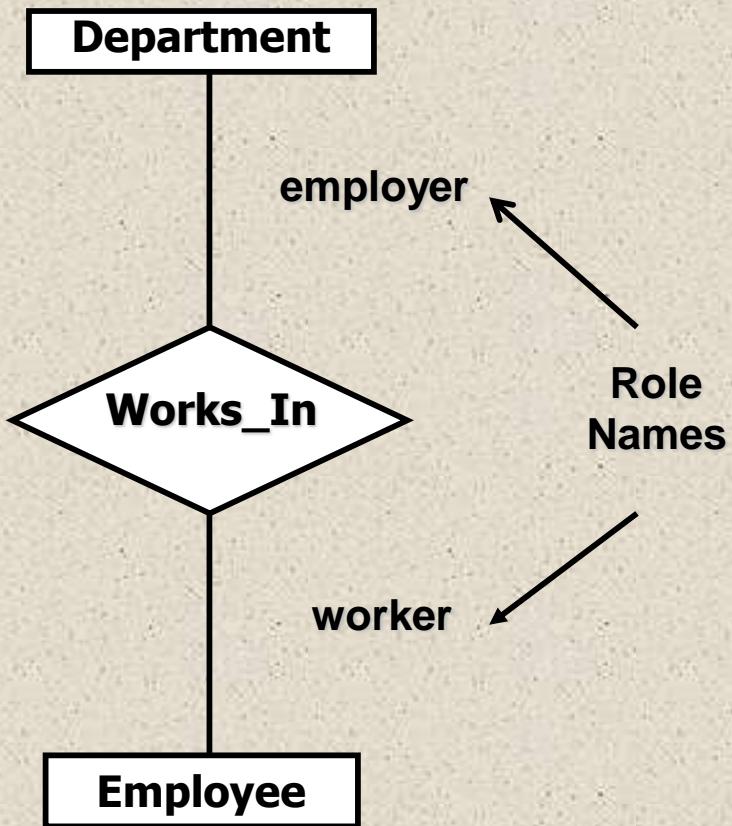


Fig 1: Role Names

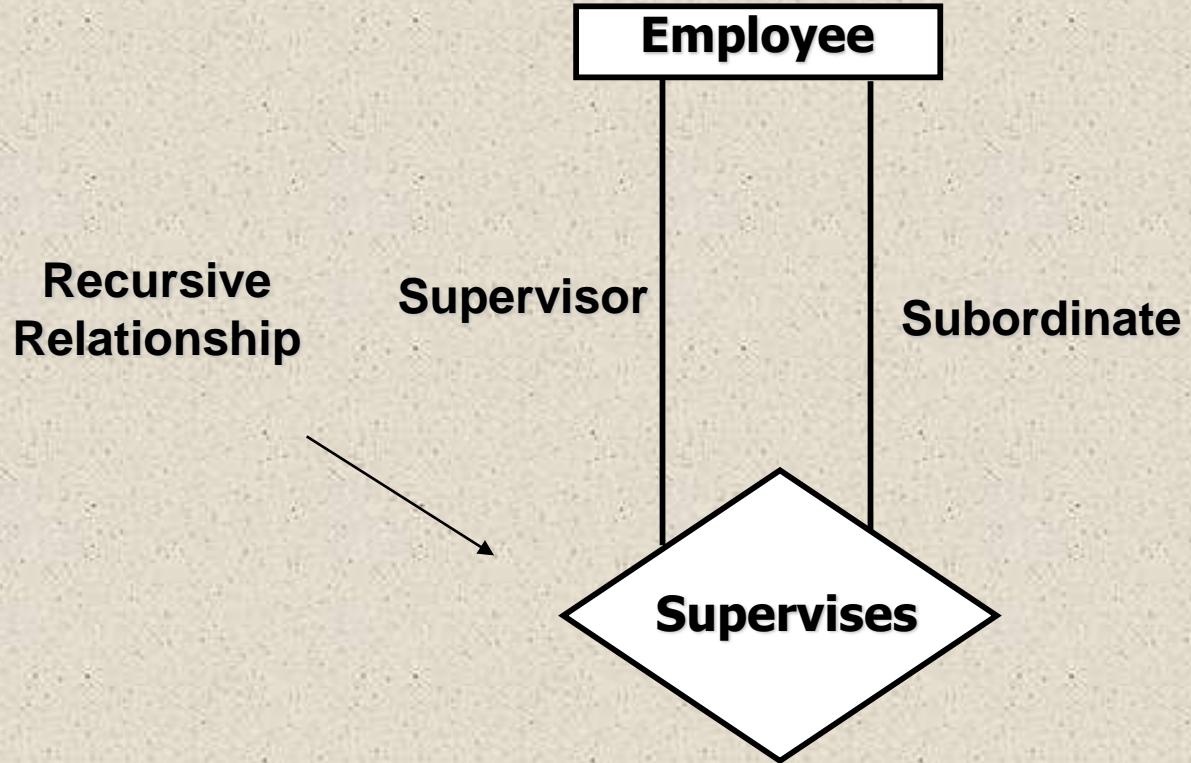


Fig 2: Recursive Relationship

The relationship in Fig. 1 has degree 2, so a binary relationship. While in Fig. 2 the degree is 1, so a unary relationship.

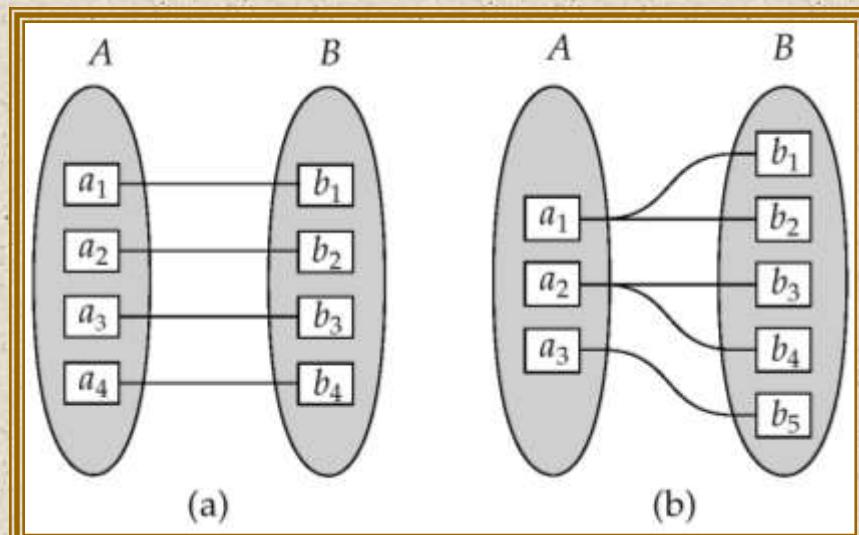
Constraints on Binary Relationship Types (Mapping / Structural Constraints)

- ◆ Relationship types have certain constraints that limit the set of possible combination of entities that may participate in the relationship type. There are mainly two types of constraints;
 - Cardinality Ratio
 - Participation Constraint
- ◆ **Cardinality Ratio:-** It specifies the **maximum number** of relationship instances that an entity can participate in.
In other words, it defines how many entities of an entity set participate in a relationship type.
Or, it express the number of entities to which another entity can be associated via a relationship set.
- ◆ Possible cardinality ratios for binary relationship types are:
 - One to one; One to many; Many to one; Many to many

Mapping Constraints (Contd.....)

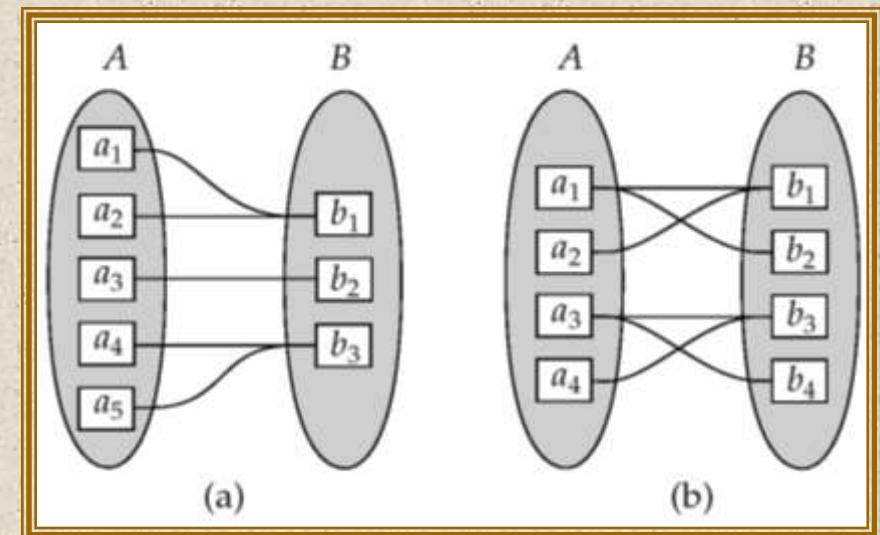
- ◆ If R is a relationship set between two entity sets A and B, then cardinality ratio can be defined as;
- ◆ **One to One**:- An entity in A is associated with at most one entity in B and an entity in B is associated with at most one entity in A.
- ◆ **One to Many**:- An entity in A is associated with any number of entities in B, while an entity in B can be associated with at most one entity in A.
- ◆ **Many to One**:- An entity in A is associated with at most one entity in B, while an entity in B can be associated with any number of entities in A.
- ◆ **Many to Many**:- An entity in A can be associated with any number of entities in B and an entity in B can be associated with any number of entities in A.

Mapping Constraints (Contd....)



One to one

One to many



Many to one

Many to many

◆ EXAMPLES :

- relationship set HAS from Department to Chairperson is one-to-one.
- relationship set TEACHES from Teacher to Student is one-to-many & TAUGHT_BY from Student to Teacher is many-to-one.
- relationship set STUDY from Student to Course is many-to-many

.

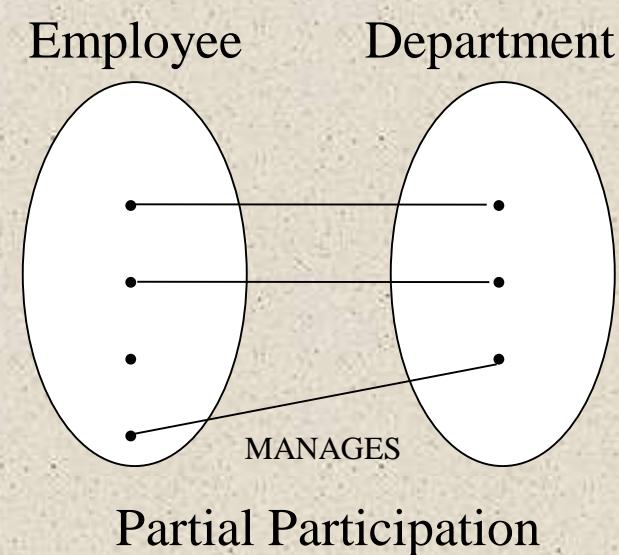
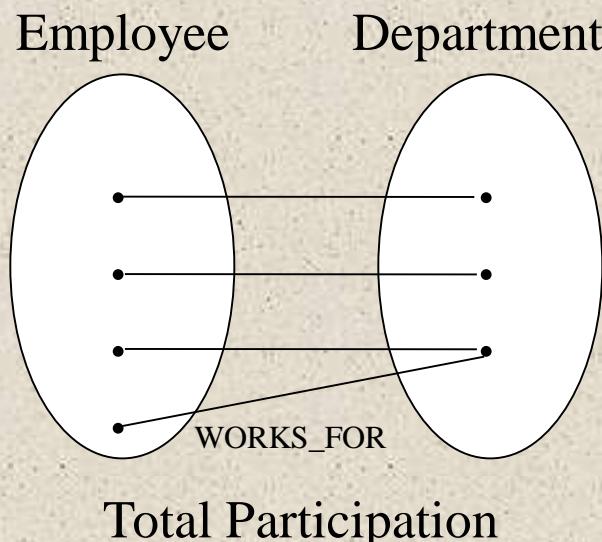
Constraints on Binary Relationship Types

(Mapping/Structural Constraints)

- ◆ **Participation Constraint:-** It describes the whether existence of an entity depends on another entity to which it is related through a relationship type
- ◆ It specifies the minimum number of relationship instances that each entity can participate in, and is sometimes called the **minimum cardinality constraint**. It may be;
 - Total Participation(Existence Dependency)
 - Partial participation
- ◆ **Total participation:-** The participation of an entity set E in a relationship set R is said to be total if each entity in E participates in at least one relationship in R i.e. each entity depends upon existence of the related entity.
- ◆ Consider Employee and Department entity sets, and a relationship WORKS_FOR between them indicating each employee must work for a department. Then there is total participation of entities from Employee entity set.

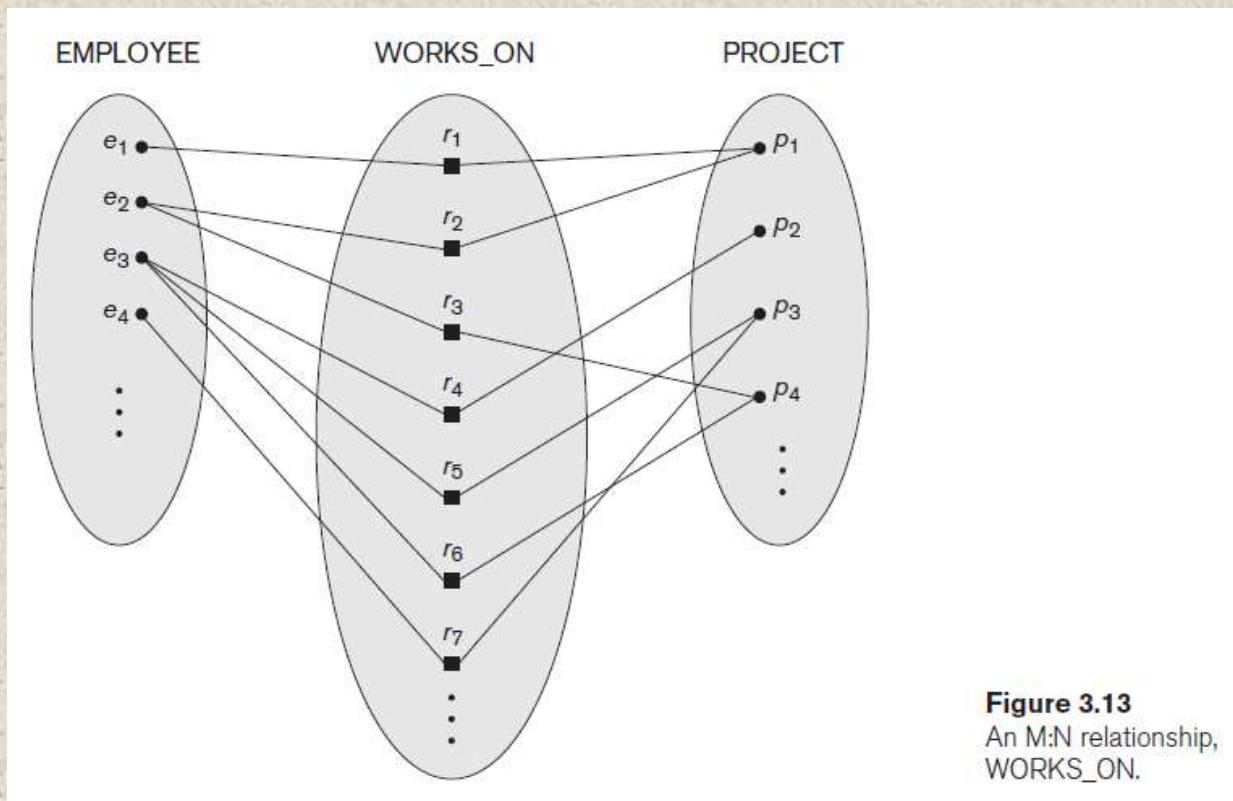
Constraints on Relationship Types (Mapping Constraints)

- ◆ **Partial participation**:- The participation of an entity set E in a relationship set R is said to be partial if only some entities in E participate in the relationships in R.
- ◆ Consider Employee and Department entity sets, and a relationship **MANAGES** between them indicating some employees manages department. Then there is partial participation of entities from Employee entity set.



Attributes of Relationship Types

- Relationship types can also have attributes, similar to those of entity types and are often known as **descriptive attributes**. 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 Figure 3.13.



Attributes of Relationship Types

- ◆ Attributes of 1:1 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

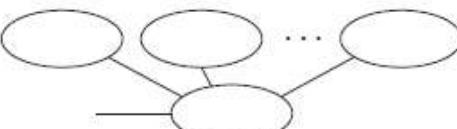
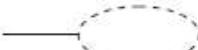
- ◆ Entity types that do not have key attributes of their own are called **weak entity types**. In contrast, **regular entity types** that do have a key attribute—which include all the examples discussed so far—are called **strong entity types**. Entities belonging to a weak entity type are identified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

Weak Entity Types

- ◆ 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. The partial key attribute is underlined with a dashed or dotted line.
- ◆ Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. The choice of which representation to use is made by the database designer.
- ◆ A weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two.
- ◆ The identifying relationship is many-to-one from the weak entity set to the identifying entity set

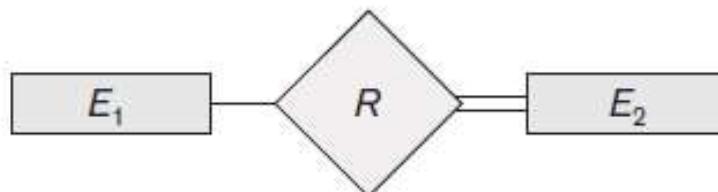
ER Diagram Notations

◆ ER- Notations:

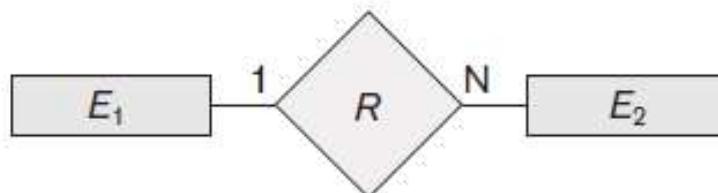
Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute

ER Diagram Notations

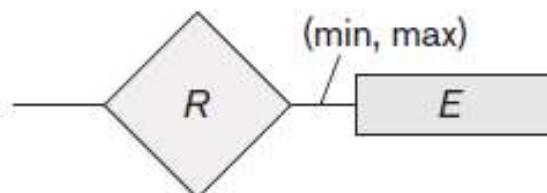
◆ ER- Notations:



Total Participation of E_2 in R

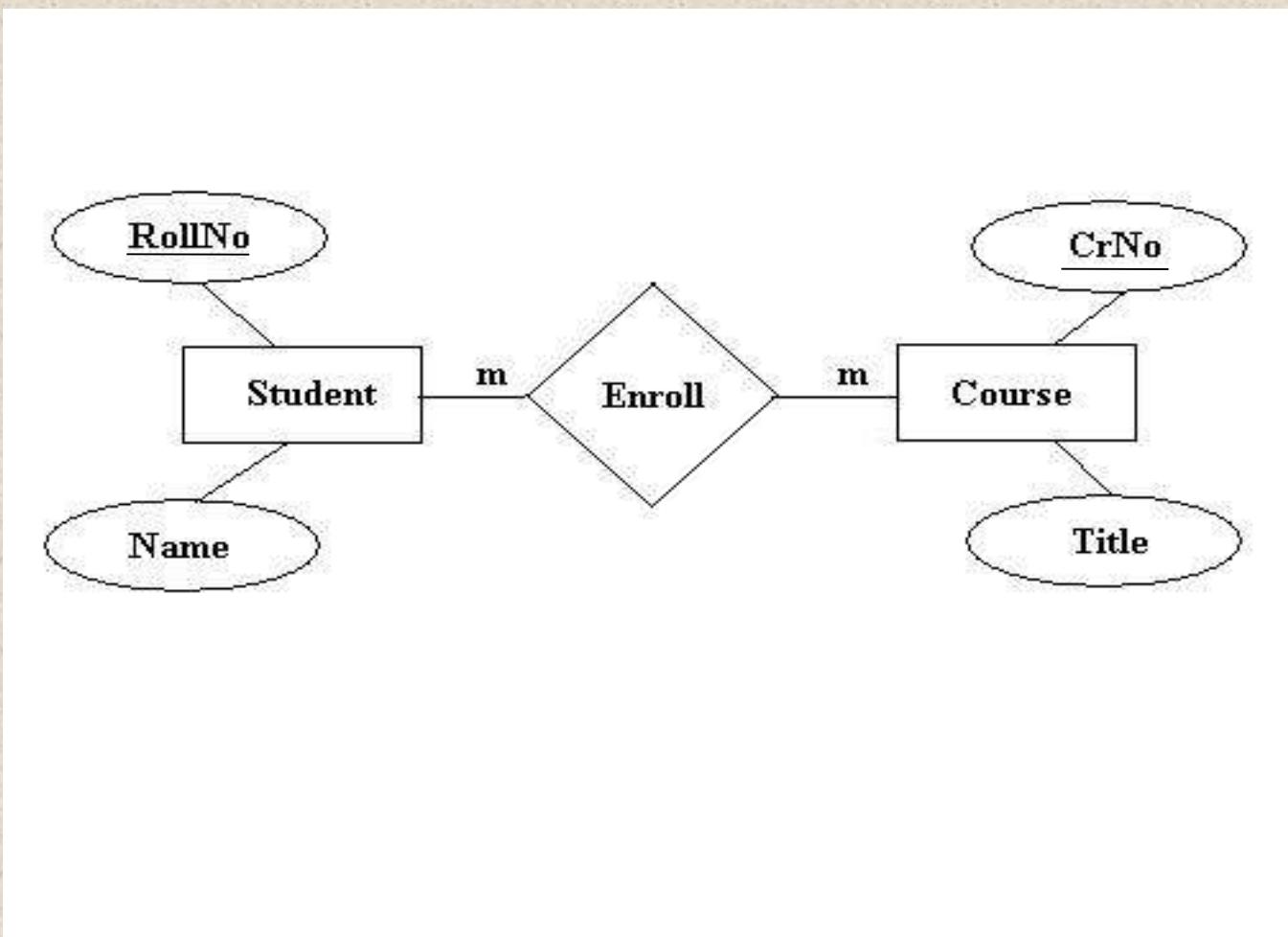


Cardinality Ratio 1: N for $E_1 : E_2$ in R

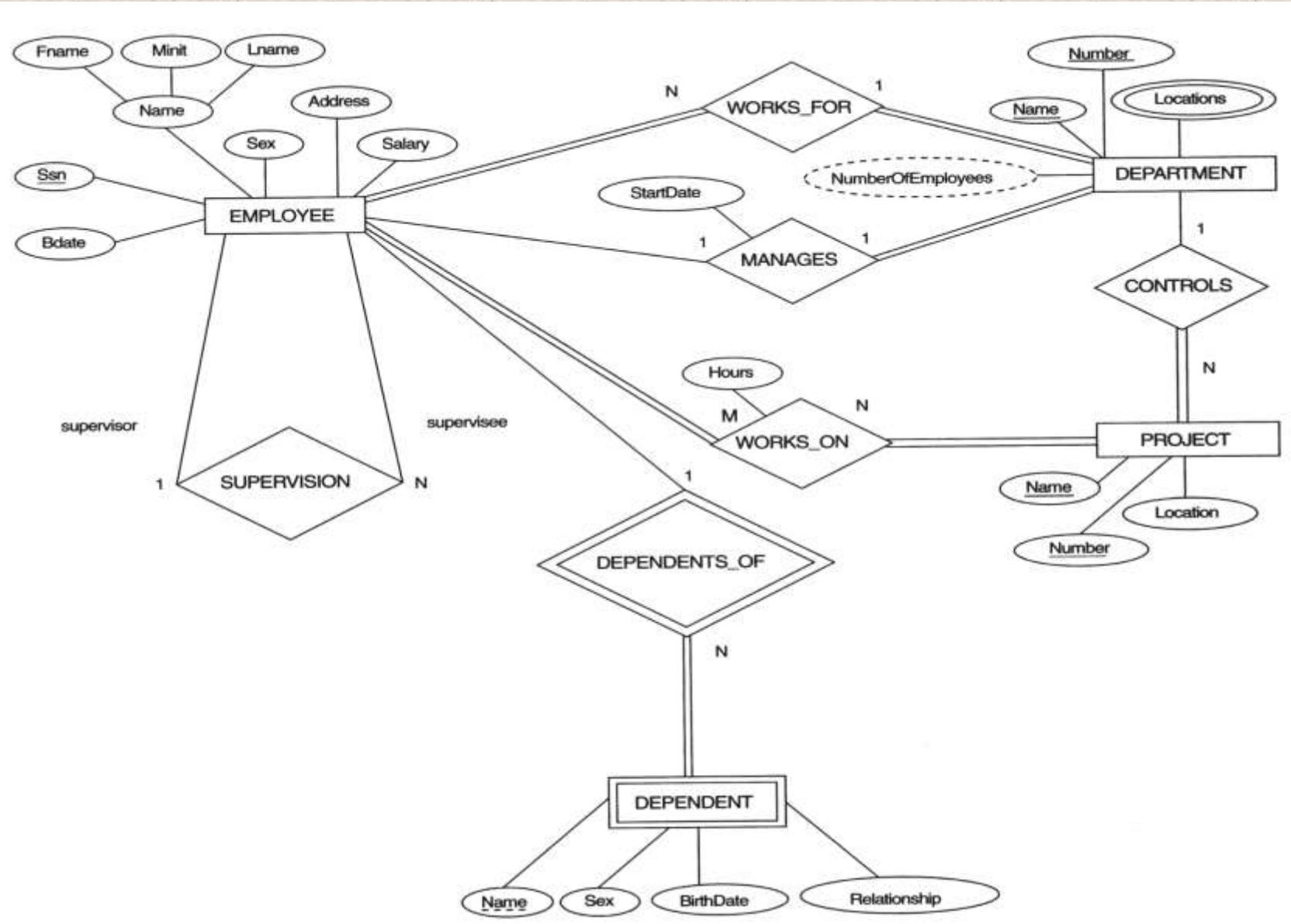


Structural Constraint (min, max)
on Participation of E in R

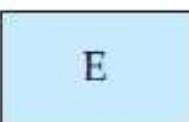
ER Diagram: An Example



ER Diagram: An Example



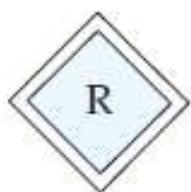
ER Diagram Notations: Based on Silberschatz Book



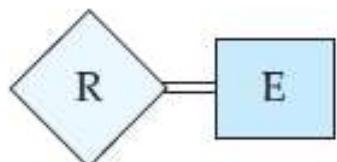
entity set



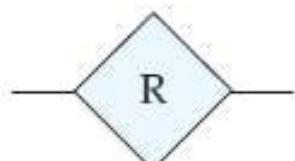
relationship set



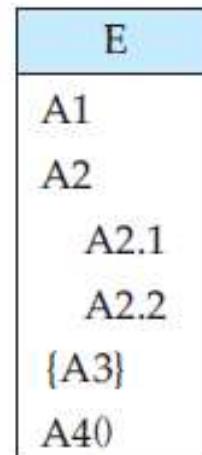
identifying
relationship set
for weak entity set



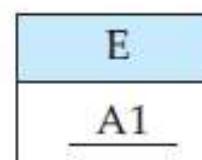
total participation
of entity set in
relationship



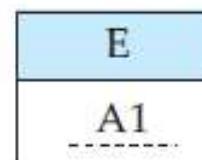
many-to-many
relationship



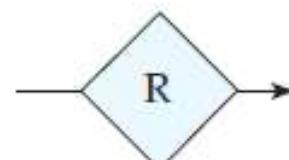
attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)



primary key

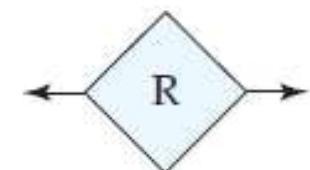


discriminating
attribute of
weak entity set

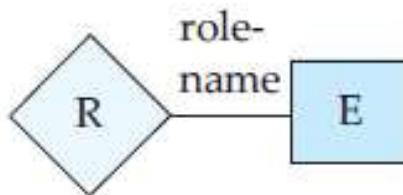


many-to-one
relationship

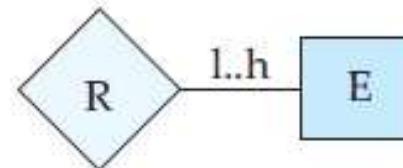
ER Diagram Notations:



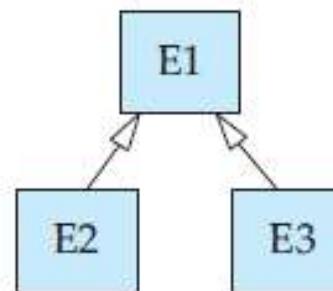
one-to-one
relationship



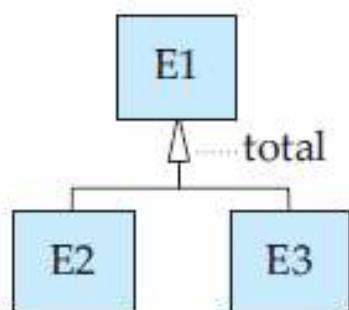
role indicator



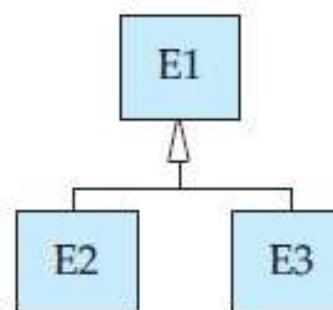
cardinality
limits



ISA: generalization
or specialization

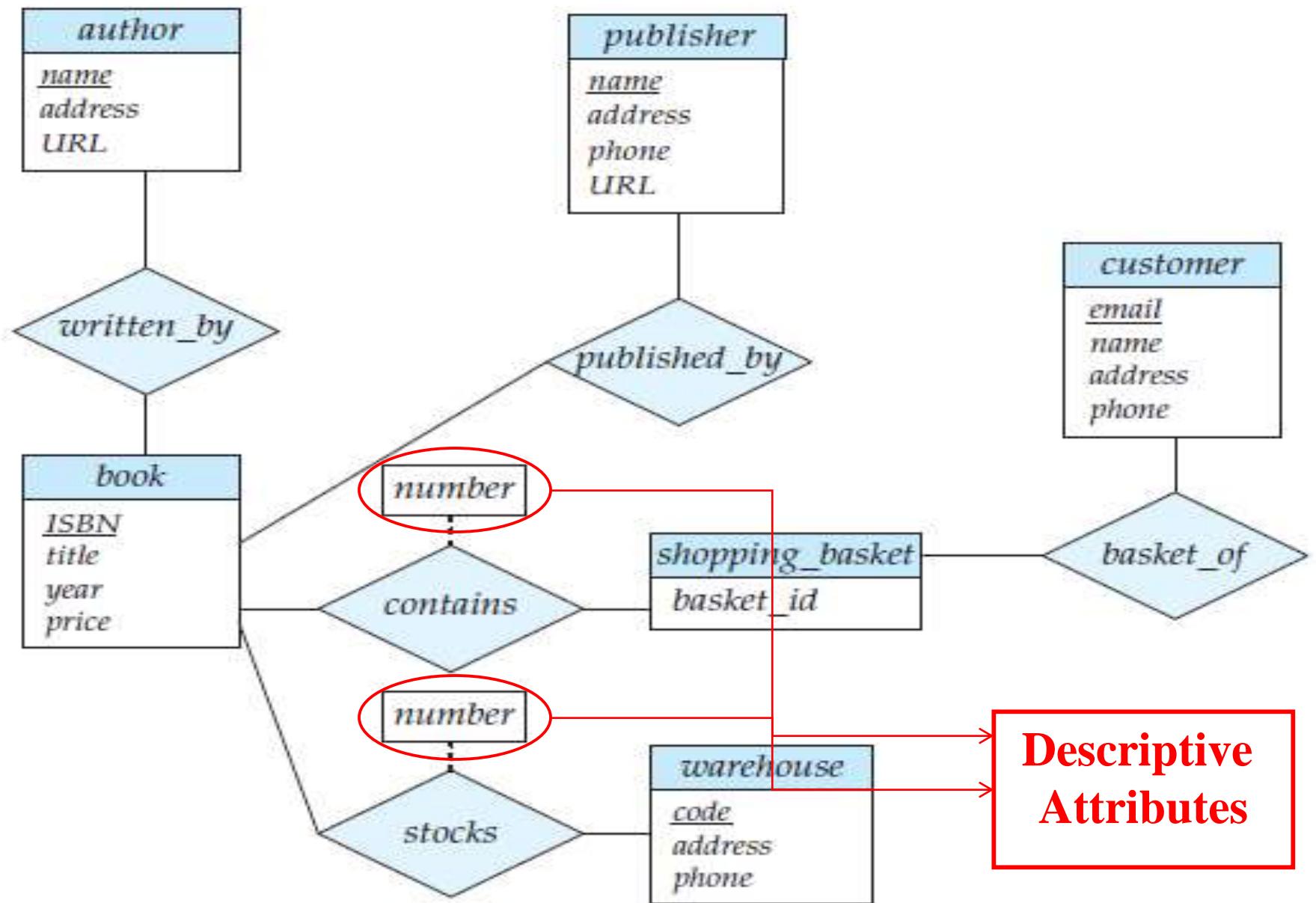


total (disjoint)
generalization



disjoint
generalization

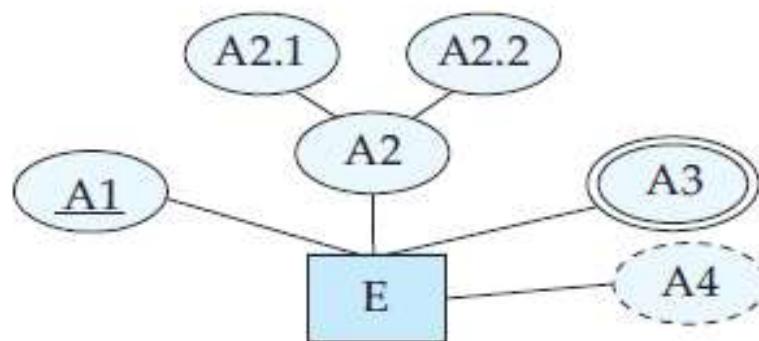
ER Diagram Notations:



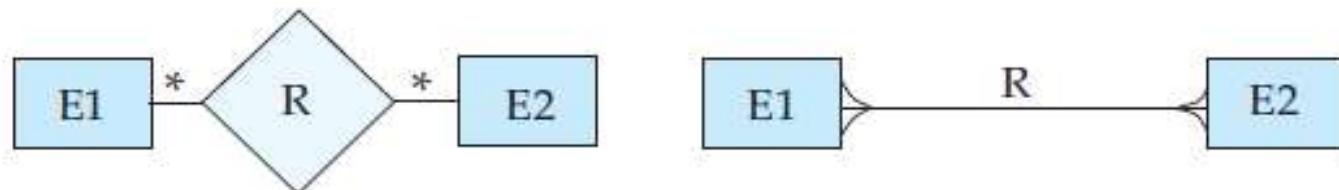
**Descriptive
Attributes**

ER Diagram Notations:

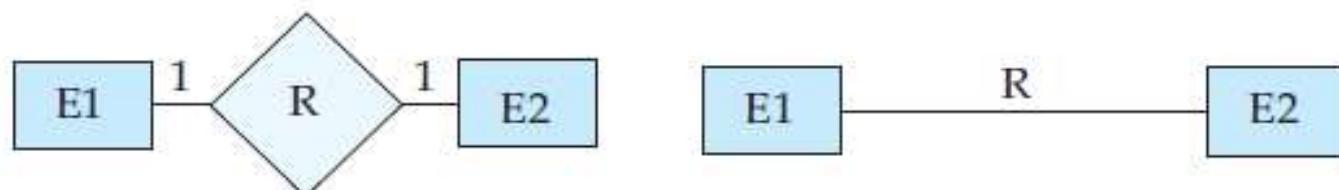
entity set E with simple attribute A1, composite attribute A2, multivalued attribute A3, derived attribute A4, and primary key A1



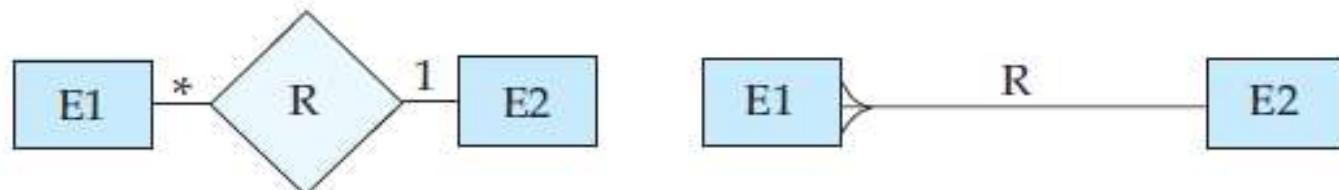
many-to-many relationship



one-to-one relationship

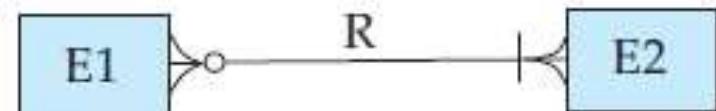
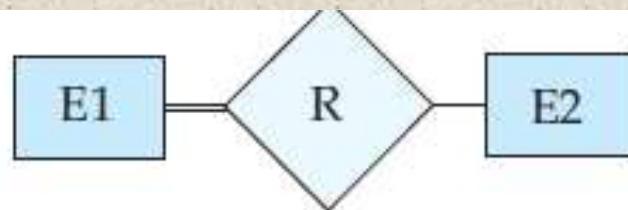


many-to-one relationship

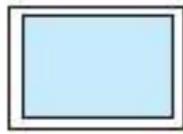


ER Diagram Notations:

participation
in R: total (E1)
and partial (E2)



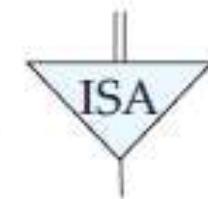
weak entity set



generalization



total
generalization



ER Diagram Notations:

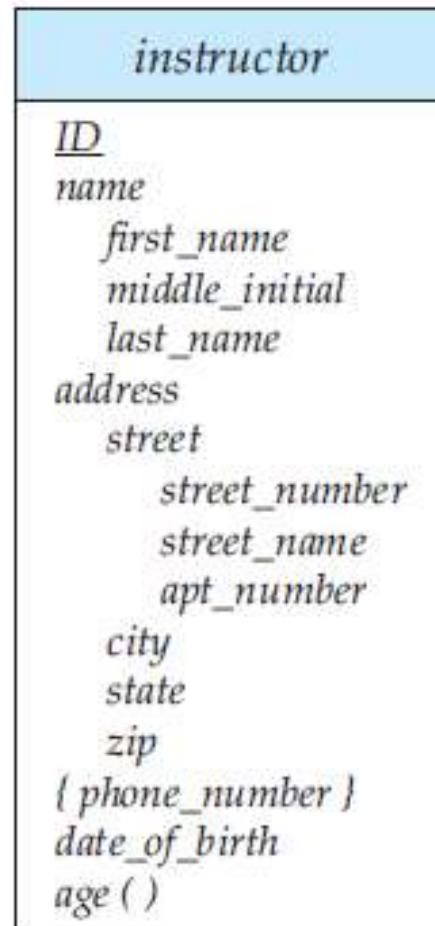
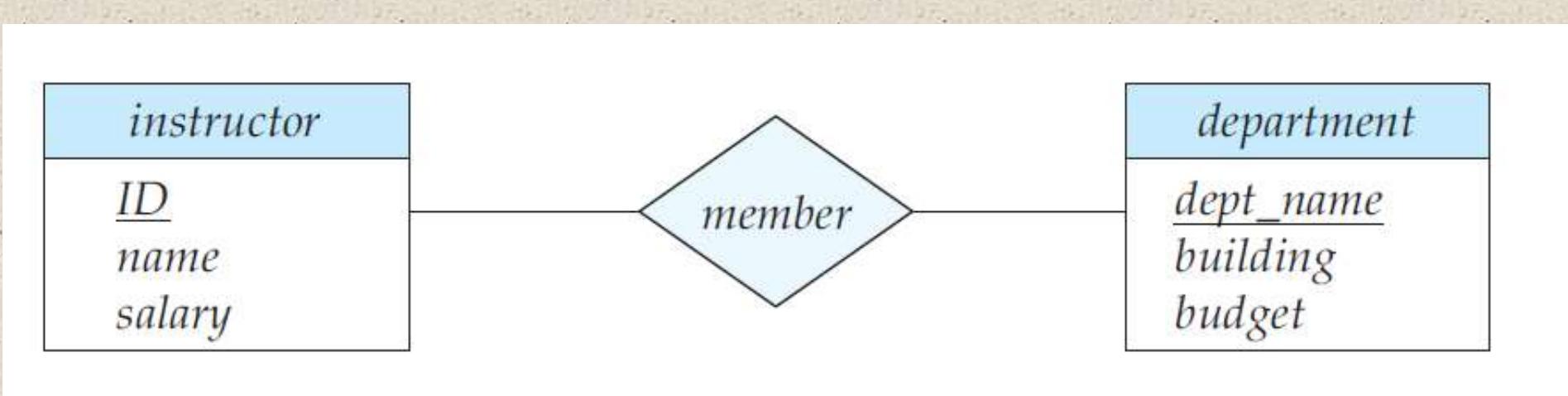
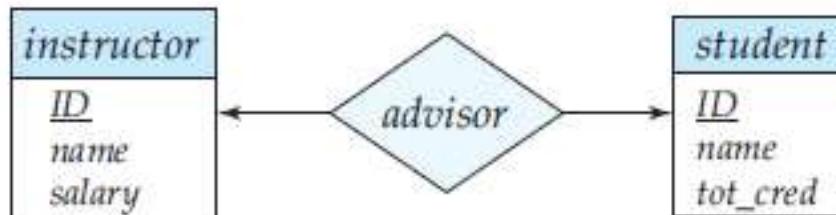


Figure 7.11 E-R diagram with composite, multivalued, and derived attributes.

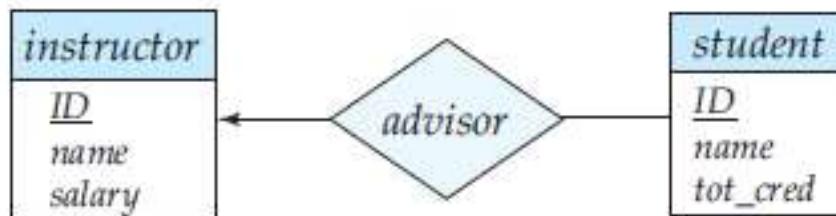
ER Diagram Notations:



ER Diagram Notations:



(a)



(b)



(c)

Figure 7.9 Relationships. (a) One-to-one. (b) One-to-many. (c) Many-to-many.

ER Diagram Notations:

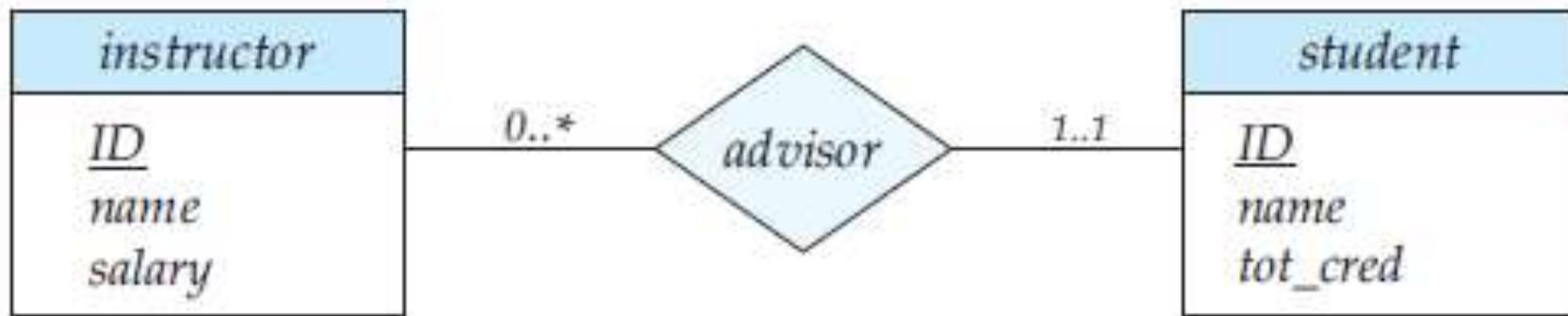


Figure 7.10 Cardinality limits on relationship sets.

ER Diagram Notations:

- ◆ Roles

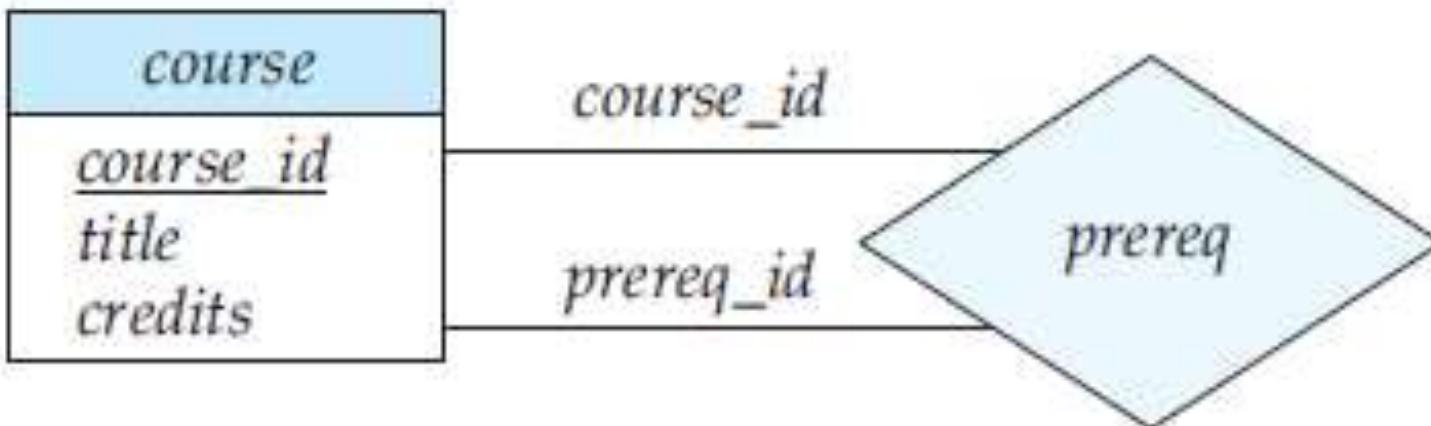


Figure 7.12 E-R diagram with role indicators.

ER Diagram Notations:

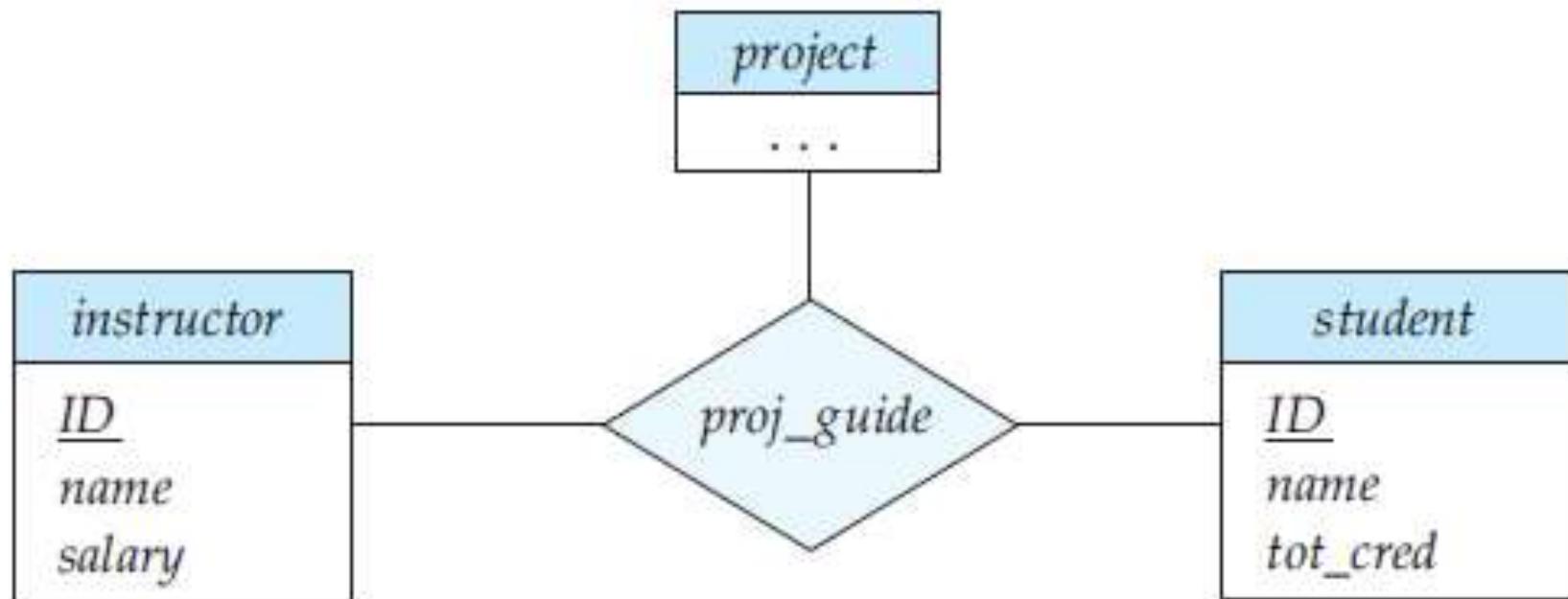
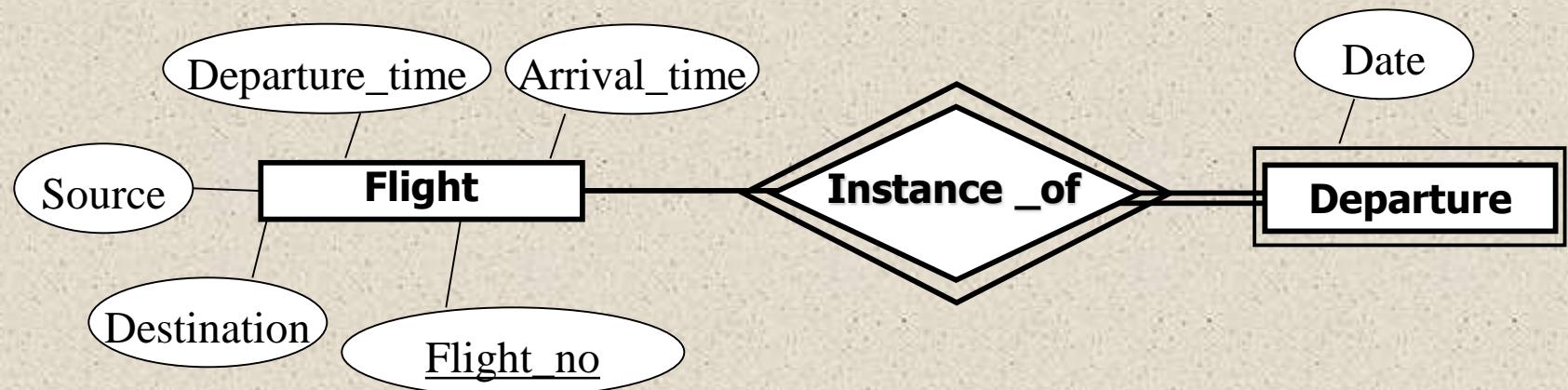


Figure 7.13 E-R diagram with a ternary relationship.

Weak Entity Example

- Here departure is a weak entity type. Even though each departure entity is distinct but different flights may have departure on same date. Thus it can be identified by relating with owner entity type “flight”, since for each flight *flight_no* is unique.



Weak Entity...

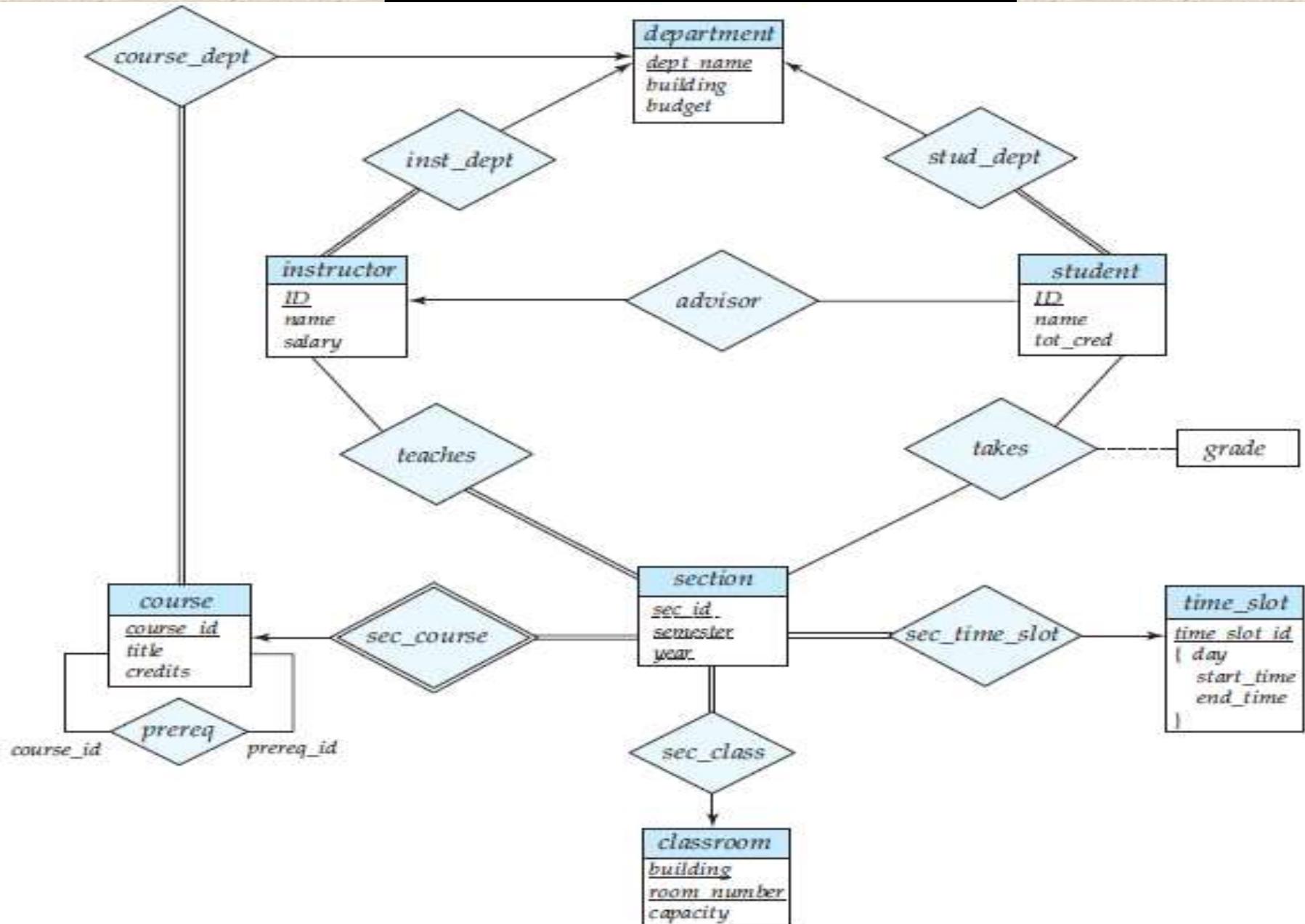
- ◆ **Partial Key:** A set of attributes that can uniquely identify weak entities that are related to the same owner entity. It is often known as **discriminator**.
- ◆ In the previous example;
 - {date} is a partial key.
 - {flight_no, date} becomes the primary key for the ***Departure*** entity.
- ◆ The entity types that do have key attributes of their own are called **strong or regular entity types**. They have their independent existence.

Weak Entity...



Figure 7.14 E-R diagram with a weak entity set.

ER Diagram for University



ER Diagram for Insurance Company

- ◆ Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

ER Diagram: Naming Convention

- ◆ When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) 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.
- ◆ Choose *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type.
- ◆ In ER diagrams, it is good to 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.

ER Diagram: Naming Convention

- ◆ As a general practice, given a narrative description of the database requirements, the *nouns* appearing in the narrative tend to give rise to entity type names, and the *verbs* tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity 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

Design Choices for ER Conceptual Design

- ◆ It is occasionally difficult to decide whether a particular concept should be modeled as an entity type, an attribute, or a relationship type.
- ◆ In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached.
- ◆ A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is often the case that a pair of such attributes that are inverses of one another are refined into a binary relationship. Once an attribute is replaced by a relationship, the attribute itself should be removed from the entity type to avoid duplication and redundancy.

Design Choices for ER Conceptual Design

- ◆ Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type. For example, suppose that each of several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE, has an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute Dept_name and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships. Other attributes/relationships of DEPARTMENT may be discovered later.
- ◆ An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute Dept_name and is related to only one other entity type, STUDENT. In this case, DEPARTMENT may be reduced or demoted to an attribute of STUDENT

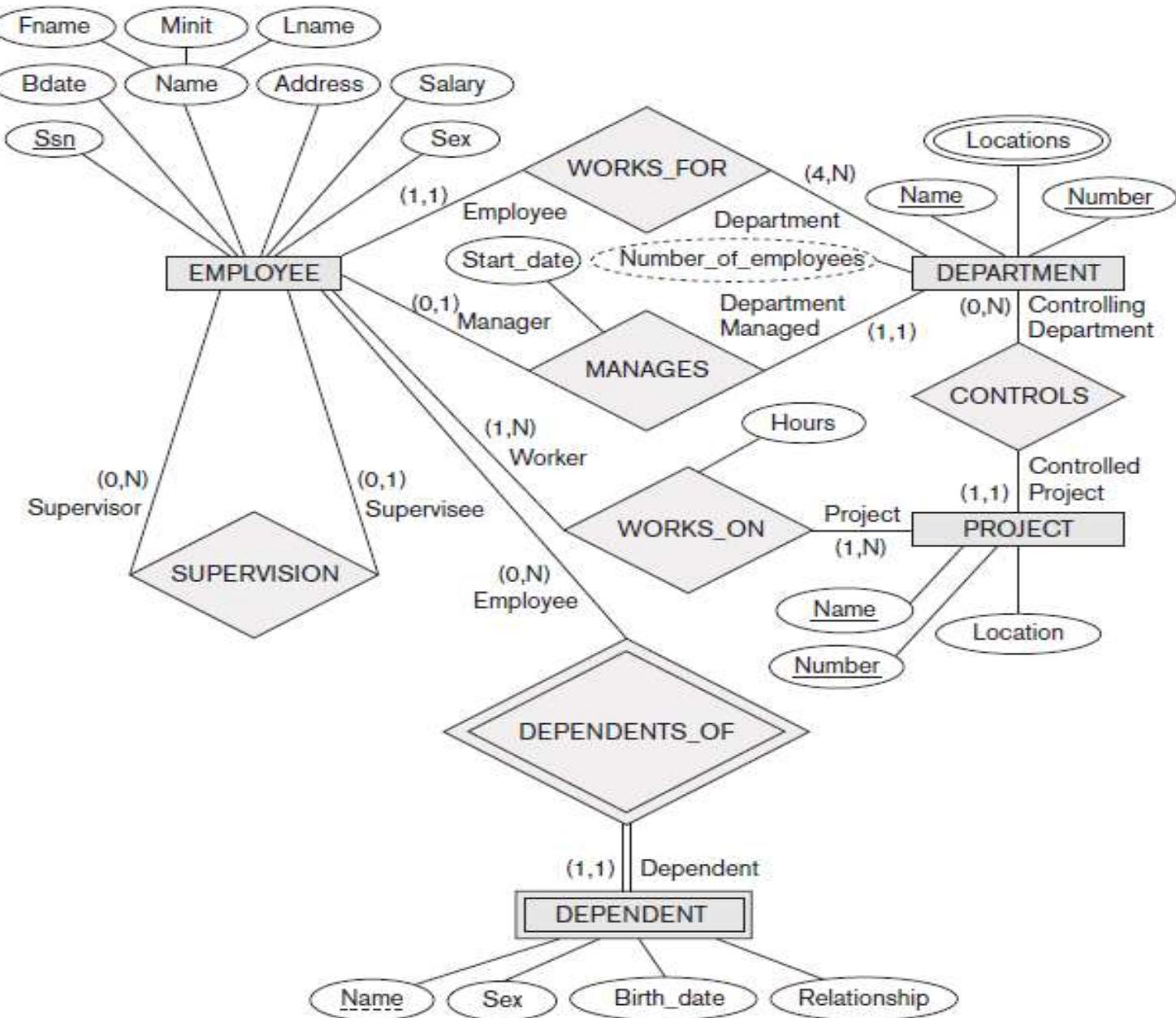


Figure 3.15

ER diagrams for the company schema, with structural constraints specified using (min, max) notation and role names.

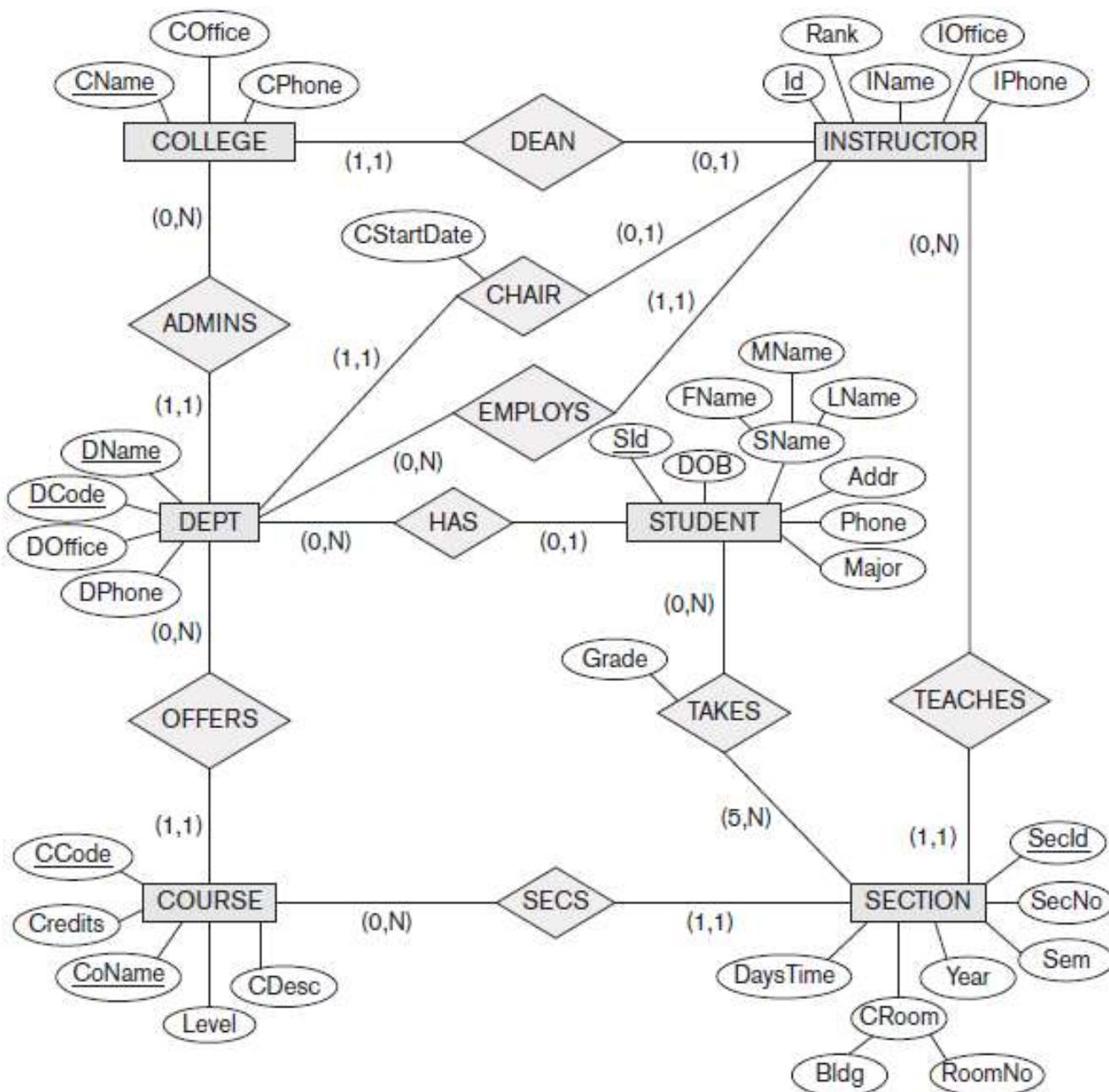


Figure 3.20

An ER diagram for a UNIVERSITY database schema.

Relationship Types of Degree Higher than Two

- ◆ We defined the **degree** of a relationship type as the number of participating entity types and called a relationship type of degree two *binary* and a relationship type of degree three *ternary*.

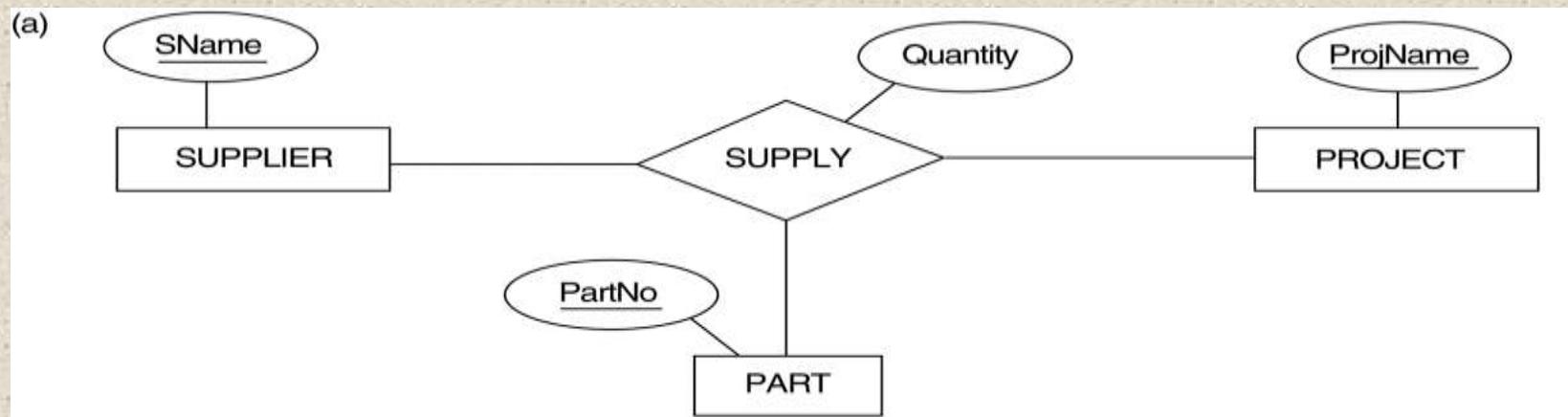
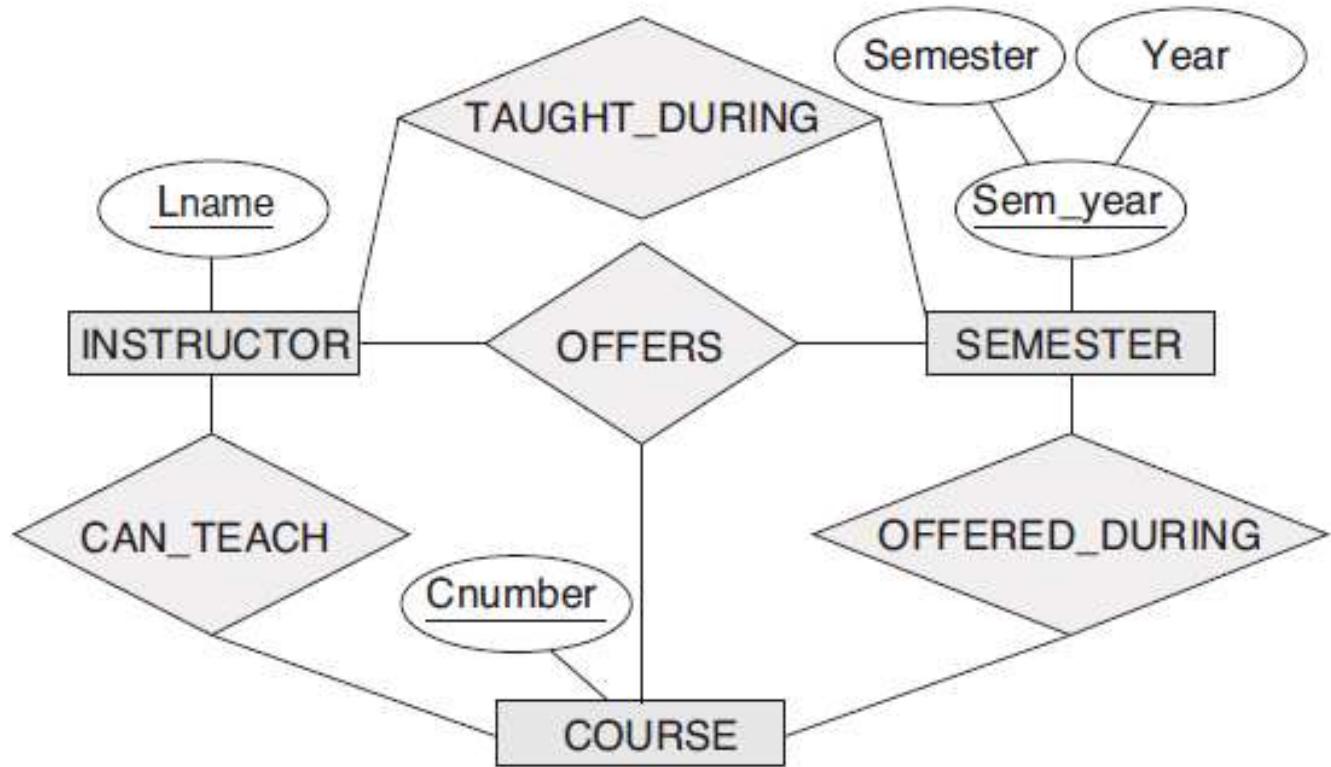


Fig: A Ternary relationship

Relationship Types of Degree Higher than Two

Figure 3.18

Another example of ternary versus binary relationship types.



Database Management System (MDS 505)

Jagdish Bhatta

Unit-2

The Relational Languages and Relational Model: Relational Model and Constraints

Relational Model

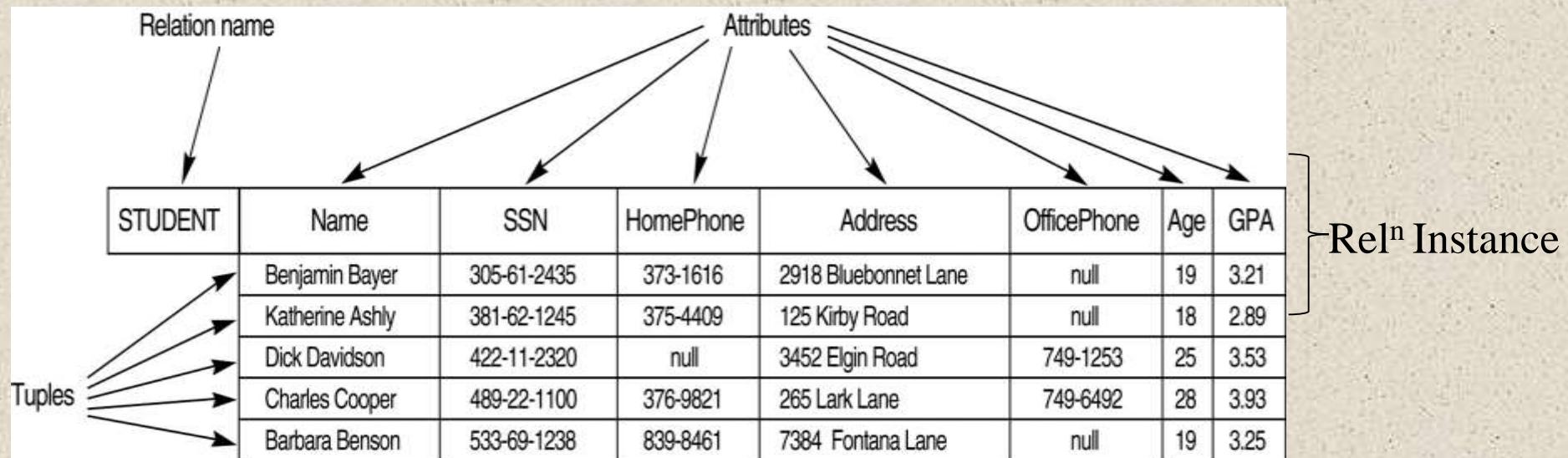
- ◆ Relational model represents the database as a collection of relations where each relation resembles a table of values with rows and columns. Major advantages of relational model over the older data models are the simple data representation & the ways complex queries can be expressed easily.
- ◆ When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

Relational Model

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

Relational Model

Eg: **Student** (Name: string, SSN: varchar, HomePhone: varchar,...) } Relⁿ Schema



Domains

- ◆ A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values.
- ◆ **Data type or format** is also specified for each domain.
- ◆ Some examples of domains follow:
 - Names: The set of character strings that represent names of persons.
 - Grade_point_averages. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.

Attributes

- ◆ A **relation schema** R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes, A_1, A_2, \dots, A_n . Each **attribute** A_i is the name of a role played by some domain D in the relation schema R . D is called the **domain** of A_i and is denoted by $\text{dom}(A_i)$. A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.
- ◆ A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

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

- ◆ Using the data type of each attribute, the definition is sometimes written as:

STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

Tuples and Relations

- ◆ **Relation** (or **relation state**) r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$. Each **n -tuple** t is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special **NULL** value.
- ◆ The i th value in tuple t , which corresponds to the attribute A_i , is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.
- ◆ Figure 5.1 in the next slide shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column. *NULL values* represent attributes whose values are unknown
- ◆ or do not exist for some individual STUDENT tuple.

Attributes, Tuples, and Relations

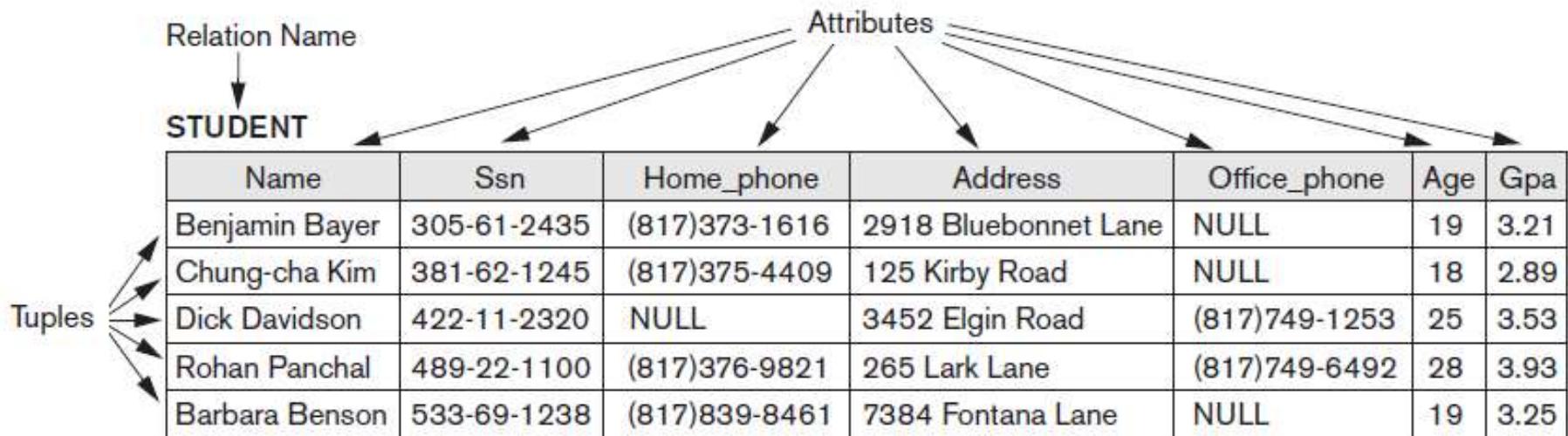


Figure 5.1

The attributes and tuples of a relation STUDENT.

Relations

- ◆ The earlier definition of a relation can be *restated* more formally using set theory concepts as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains $\text{dom}(A1), \text{dom}(A2), \dots, \text{dom}(An)$, which is a **subset** of the **Cartesian product** (denoted by \times) of the domains that define R :

$$r(R) \subseteq (\text{dom}(A1) \times \text{dom}(A2) \times \dots \times \text{dom}(An))$$

- ◆ The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality**, in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|\text{dom}(A1)| \times |\text{dom}(A2)| \times \dots \times |\text{dom}(An)|$$

- ◆ This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. Of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real world changes, so does the relation state, by being transformed into another relation state. However, the schema R is relatively static and changes *very infrequently*—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

Characteristics of relations:

- ◆ **Ordering of Tuples in a Relation:** A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order. Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level.
- ◆ **Ordering of Values within a Tuple and an Alternative Definition of a Relation:** A relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a *set* of attributes (instead of an ordered list of attributes), and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \dots, t_m\}$, where each tuple t_i is a **mapping** from R to D , and D is the **union** (denoted by \cup) of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r . Each mapping t_i is called a tuple. According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of (**attribute**, **value**) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$.

Characteristics of relations:

- ◆ **Values and NULLs in the Tuples:** Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. In general, we can have several meanings for NULL values, such as *value unknown*, *value exists but is not available*, or *attribute does not apply* to this tuple (also known as *value undefined*).
- ◆ **Interpretation (Meaning) of a Relation.** The relation schema can be interpreted as a declaration or a type of **assertion**. For example, the schema of the STUDENT relation of Figure 5.1, in slide number 9, asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. For example, the first tuple in Figure 5.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on. An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. For example, the predicate STUDENT (Name, Ssn, ...) is true for the five tuples in relation STUDENT of Figure 5.1.

Relational Model Notations

- ◆ A relation schema R of degree n is denoted by $R(A_1, A_2, \dots, A_n)$.
- ◆ The uppercase letters Q, R, S denote relation names.
- ◆ The lowercase letters q, r, s denote relation states.
- ◆ The letters t, u, v denote tuples.
- ◆ In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation—the *current relation state*—whereas STUDENT(Name, Ssn, ...) refers *only* to the relation schema.
- ◆ An attribute A can be qualified with the relation name R to which it belongs by using the dot notation $R.A$ —for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.

Relational Model Notations

- ◆ An n -tuple t in a relation $r(R)$ is denoted by $t = \langle v1, v2, \dots, vn \rangle$, where vi is the value corresponding to attribute Ai . The following notation refers to **component values** of tuples:
 - Both $t[Ai]$ and $t.Ai$ (and sometimes $t[i]$) refer to the value vi in t for attribute Ai .
 - Both $t[Au, Aw, \dots, Az]$ and $t.(Au, Aw, \dots, Az)$, where Au, Aw, \dots, Az is a list of attributes from R , refer to the subtuple of values $\langle vu, vw, \dots, vz \rangle$ from t corresponding to the attributes specified in the list.
- ◆ As an example, consider the tuple $t = \langle \text{'Barbara Benson'}, \text{'533-69-1238'}, \text{'(817)839-8461'}, \text{'7384 Fontana Lane'}, \text{NULL}, 19, 3.25 \rangle$ from the STUDENT relation in Figure 5.1, in slide number 9; we have $t[\text{Name}] = \langle \text{'Barbara Benson'} \rangle$, and $t[\text{Ssn, Gpa, Age}] = \langle \text{'533-69-1238'}, 3.25, 19 \rangle$.

Relational Model Constraints

- ◆ There are generally many restrictions or **constraints** on the actual values in a database state.
- ◆ Constraints on databases can generally be divided into three main categories:
 - Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
 - Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL. We call these **schema-based constraints** or **explicit constraints**.
 - Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

Relational Model Constraints

- ◆ There are generally many restrictions or **constraints** on the actual values in a database state.
- ◆ Constraints on databases can generally be divided into three main categories:
 - Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
 - Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL. We call these **schema-based constraints** or **explicit constraints**.
 - Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

Relational Model Constraints

- ◆ The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.
- ◆ **Domain Constraints:** Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double-precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and other special data types. Domains can also be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed.

Relational Model Constraints

- ◆ **Key Constraints and Constraints on NULL Values:** In the formal relational model, a *relation* is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct.
- ◆ This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples t_1 and t_2 in a relation state r of R , we have the constraint that:

$$t_1[\text{SK}] \neq t_2[\text{SK}].$$

Relational Model Constraints

- ◆ Any such set of attributes SK is called a **superkey** of the relation schema R . A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state r of R can have the same value for SK . Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy. A **key** k of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more. Hence, a key satisfies two properties:
 - Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This *uniqueness* property also applies to a superkey.
 - It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold. This *minimality* property is required for a key but is optional for a superkey.

Relational Model Constraints

- ◆ Any such set of attributes SK is called a **superkey** of the relation schema R . A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state r of R can have the same value for SK . Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a **key**, which has no redundancy. A **key** k of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more. Hence, a key satisfies two properties:
 - Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This *uniqueness* property also applies to a superkey.
 - It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold. This *minimality* property is required for a key but is optional for a superkey.

Relational Model Constraints

- ◆ Hence, a key is a superkey but not vice versa. A superkey may be a key (if it is minimal) or may not be a key (if it is not minimal). Consider the STUDENT relation of Figure 5.1, in slide number 9. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.⁸ Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes together to have the uniqueness property.

Relational Model Constraints

- ◆ The value of a key attribute can be used to identify uniquely each tuple in the relation.
- ◆ Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema.
- ◆ A key is determined from the meaning of the attributes, and the property is *time-invariant*: It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 5.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.

Relational Model Constraints

- ◆ In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 5.4, in slide number 23, 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. This is the candidate key whose values are used to *identify* tuples in the relation.
- ◆ The convention that the attributes that form the primary key of a relation schema are underlined is followed in general.
- ◆ When a relation schema has several candidate keys, the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys** and are not underlined

Relational Model Constraints

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4

The CAR relation, with two candidate keys:
License_number and
Engine_serial_number.

Relational Model Constraints

- ◆ Another constraint on attributes specifies whether NULL values are or are not permitted.
- ◆ For example, if every STUDENT tuple must have a valid, non-NUL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

Relational Databases and Relational Database Schemas

- ◆ A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state** DB of S is a set of relation states $\text{DB} = \{r_1, r_2, \dots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified. Figure 5.5, in slide number 27, shows a relational database schema that we call $\text{COMPANY} = \{\text{EMPLOYEE}, \text{DEPARTMENT}, \text{DEPT_LOCATIONS}, \text{PROJECT}, \text{WORKS_ON}, \text{DEPENDENT}\}$. In each relation schema, the underlined attribute represents the primary key. The figure 5.6 shows a relational database state corresponding to the COMPANY schema.
- ◆ When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called **not valid**, and a state that satisfies all the constraints in the defined set of integrity constraints is called a **valid state**.

Relational Databases and Relational Database Schemas

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.

Relational Databases and Relational Database Schemas

- ◆ In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation of Figure 5.5, in previous slide: once in the role of the employee's SSN, and once in the role of the supervisor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

Relationship Schemas

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

Entity Integrity, Referential Integrity, and Foreign Keys

- ◆ The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.
- ◆ Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 5.6, in the previous slide, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

Entity Integrity, Referential Integrity, and Foreign Keys

- ◆ To define *referential integrity* more formally, first we define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2 . A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:
 - The attributes in FK have the same domain(s) as the primary key attributes PK of R_2 ; the attributes FK are said to **reference** or **refer to** the relation R_2 .
 - A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is $NULL$. In the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2 .
- ◆ In this definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

Entity Integrity, Referential Integrity, and Foreign Keys

- ◆ Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 5.6, slide number 29. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of Dno *can be NULL* if the employee does not belong to a department or will be assigned to a department later. For example, in Figure 5.6, in slide number 29, the tuple for employee ‘John Smith’ references the tuple for the ‘Research’ department, indicating that ‘John Smith’ works for this department.

Entity Integrity, Referential Integrity, and Foreign Keys

- ◆ Notice that a foreign key can *refer to its own relation*. For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure 5.6, in the slide number 29, the tuple for employee ‘John Smith’ references the tuple for employee ‘Franklin Wong,’ indicating that ‘Franklin Wong’ is the supervisor of ‘John Smith’.
- ◆ We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 5.7, in next slide, shows the schema in Figure 5.5, in slide number 27, with the referential integrity constraints displayed in this manner.

Entity Integrity, Referential Integrity, and Foreign Keys

Figure 5.7

Referential integrity constraints displayed on the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

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

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	Dependent_name	Sex	Bdate	Relationship
-------------	----------------	-----	-------	--------------

Entity Integrity, Referential Integrity, and Foreign Keys

- ◆ All integrity constraints should be specified on the relational database schema (that is, specified as part of its definition) if we want the DBMS to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them.
- ◆ In SQL, the CREATE TABLE statement of the SQL DDL allows the definition of primary key, unique key, NOT NULL, entity integrity, and referential integrity constraints, among other constraints

Other Types of Constraints

- ◆ The preceding integrity constraints are included in the data definition language because they occur in most database applications. Another class of general constraints, sometimes called *semantic integrity constraints*, are not part of the DDL and have to be specified and enforced in a different way. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee's supervisor* and *the maximum number of hours an employee can work on all projects per week is 56*. Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**. Mechanisms called **triggers** and **assertions** can be used in SQL, through the CREATE ASSERTION and CREATE TRIGGER statements, to specify some of these constraints. It is more common to check for these types of constraints within the application programs than to use constraint specification languages because the latter are sometimes difficult and complex to use.

Other Types of Constraints

- ◆ The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy.
- ◆ Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database. An example of a transition constraint is: “the salary of an employee can only increase.” Such constraints are typically enforced by the application programs or specified using active rules and triggers.

Database Modification and Dealing with Constraint Violations

- ◆ There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records, respectively.
- ◆ **Insert** is used to insert one or more new tuples in a relation.
- ◆ **Delete** is used to delete tuples.
- ◆ **Update (or Modify)** is used to change the values of some attributes in existing tuples.
- ◆ Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated.

Database Modification and Dealing with Constraint Violations

- ◆ **The 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.
 - Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type.
 - Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation $r(R)$.
 - Entity integrity can be violated if any part of the primary key of the new tuple t is NULL. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation.

Database Modification and Dealing with Constraint Violations

- ◆ **The Insert Operation:**
- ◆ If an insertion violates one or more constraints, the default option is to *reject the insertion*. In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected.

Database Modification and Dealing with Constraint Violations

- ◆ **The Delete Operation:**
- ◆ The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted.
- ◆ Several options are available if a deletion operation causes a violation. The first option, called **restrict**, is to *reject the deletion*. The second option, called **cascade**, is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. A third option, called **set null** or **set default**, is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple. Notice that if a referencing attribute that causes a violation is *part of the primary key*, it *cannot* be set to NULL; otherwise, it would violate entity integrity.

Database Modification and Dealing with Constraint Violations

- ◆ **The Update Operation:**
- ◆ The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) 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.
- ◆ Updating an attribute that is *neither part of a primary key nor part of a foreign key* usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples. Hence, the issues discussed earlier for Insert and Delete come into play.

Relational Database Design Using ER-to-Relational Mapping

- ◆ It includes how to *design a relational database schema* based on a conceptual schema design. Here, we focus on the **logical database design** step of database design, which is also known as **data model mapping**. We present the procedures to create a relational schema from an entity–relationship (ER).
- ◆ It includes a seven-step algorithm to convert the basic ER model constructs—entity types (strong and weak), binary relationships (with various structural constraints), n -ary relationships, and attributes (simple, composite, and multivalued)—into relations

Reduction of E-R Schema to Tables

- ◆ As a part of database design, representing the logical structure of database with the relational tables needs conversion of the conceptual ER-representation into the tables. This conversion is the mapping of ER-schema into the tables.
- ◆ Reduction of E-R schema into tables include following algorithmic steps:

Step 1: Mapping of Regular Entity Types

Step 2: Mapping of Weak Entity Types

Step 3: Mapping of Binary 1:1 Relation Types

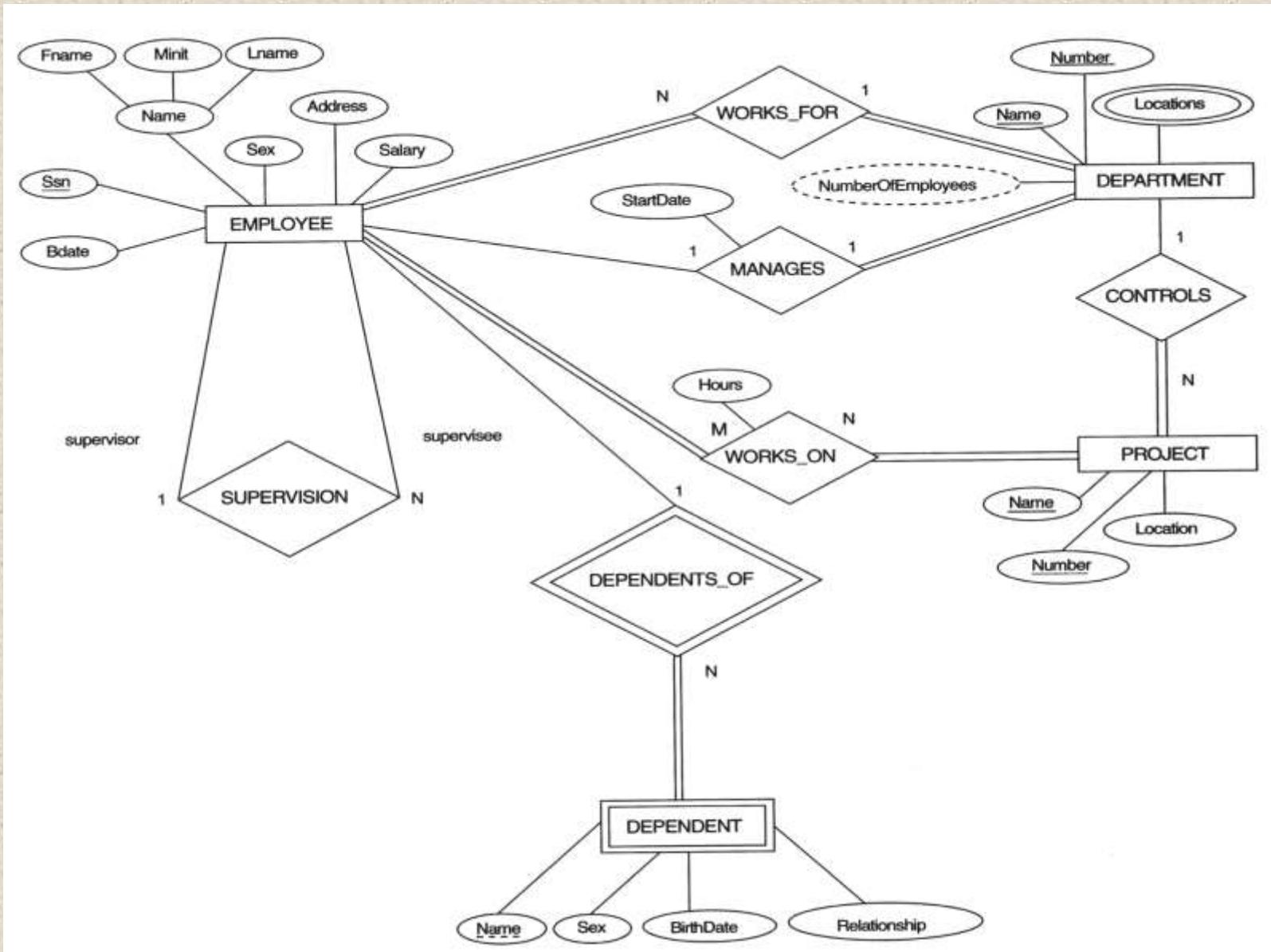
Step 4: Mapping of Binary 1:N Relationship Types.

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

Step 6: Mapping of Multivalued attributes.

Step 7: Mapping of N-ary Relationship Types.

ER to Relational Mapping: An Example



Step1: Mapping of Regular Entity Types:

- ◆ For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E.
- ◆ Include only simple component attributes of a composite attribute.
- ◆ Choose one of the key attributes of E as the primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.

Step2: 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 attributes (or simple components of composite attributes) of W as attributes of R .
- ◆ In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).
- ◆ The primary key of R is the *combination* of the primary key(s) of the owner(s) and the partial key of the weak entity type W , if any. If there is a weak entity type $E2$ whose owner is also a weak entity type $E1$, then $E1$ should be mapped before $E2$ to determine its primary key first.

Step3: 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) Foreign Key approach: Choose one of the relations, say, S and include the primary key of T as a foreign key in S. To choose role of S, it is better to choose an entity type with *total participation* in the relationship R. Include attributes of the relationship type, if any, as attributes of S. This approach is the most useful and should be followed unless special conditions exist.

Example: 1:1 relation MANAGES is mapped by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total. We include the 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 type in the DEPARTMENT relation and rename it Mgr_start_date

Step3: Mapping of binary 1:1 Relationship Types:

(2) **Merged relation option:** An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when *both participations are total*.

(3) **Cross-reference or relationship relation option:** The third alternative 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**), because each tuple in R represents a relationship instance that relates one tuple from S with one tuple from T . The relation R will include the primary key attributes of S and T as foreign keys to S and T . The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R . The drawback is having an extra relation, and requiring extra join operations when combining related tuples from the tables.

Step4: Mapping of binary 1:N Relationship Types:

- ◆ There are two possible approaches: (1) the foreign key approach and (2) the cross-reference or relationship relation approach. The first approach is generally preferred as it reduces the number of tables.
- ◆ **The foreign key approach:** For each regular binary 1:N relationship type R , identify the relation S that represents the 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 represents the other entity type participating in R ; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S .

Step4: Mapping of binary 1:N Relationship Types:

- ◆ **The relationship relation approach:** An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation R whose attributes are the primary keys of S and T , which will also be foreign keys to S and T . The primary key of R is the same as the primary key of S . This option can be used if few tuples in S participate in the relationship to avoid excessive NULL values in the foreign key.
- ◆ For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side.

Step5: Mapping of binary M:N Relationship Types:

- ◆ In the traditional relational model with no multivalued attributes, the only option for M:N relationships is the **relationship relation (cross-reference) option**. 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 (or simple components of composite attributes) as attributes of S . Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate *relationship relation* S .

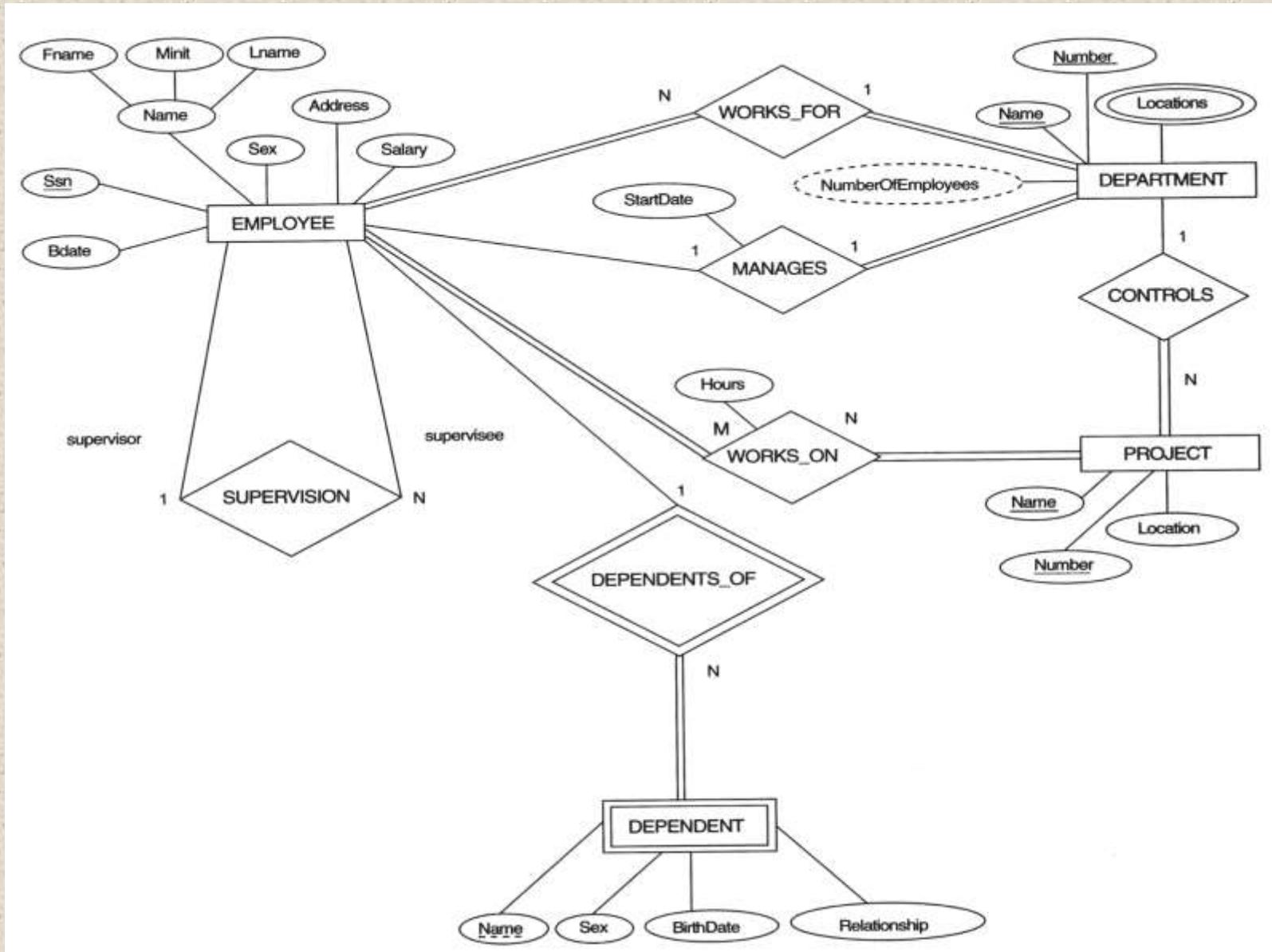
Step6: Mapping of Multivalued Attributes:

- ◆ For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, as well as the primary key attribute K of the relation that represents the entity type or relationship type that has A as an attribute as a foreign key in R.
- ◆ The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.

Step7: Mapping of N-ary Relationship Types:

- ◆ We use the **relationship relation option**. 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 the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n -ary relationship type (or simple components of composite attributes) as attributes of S . The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E' corresponding to E

ER to Relational Mapping: An Example



ER to Relational Mapping: An Example

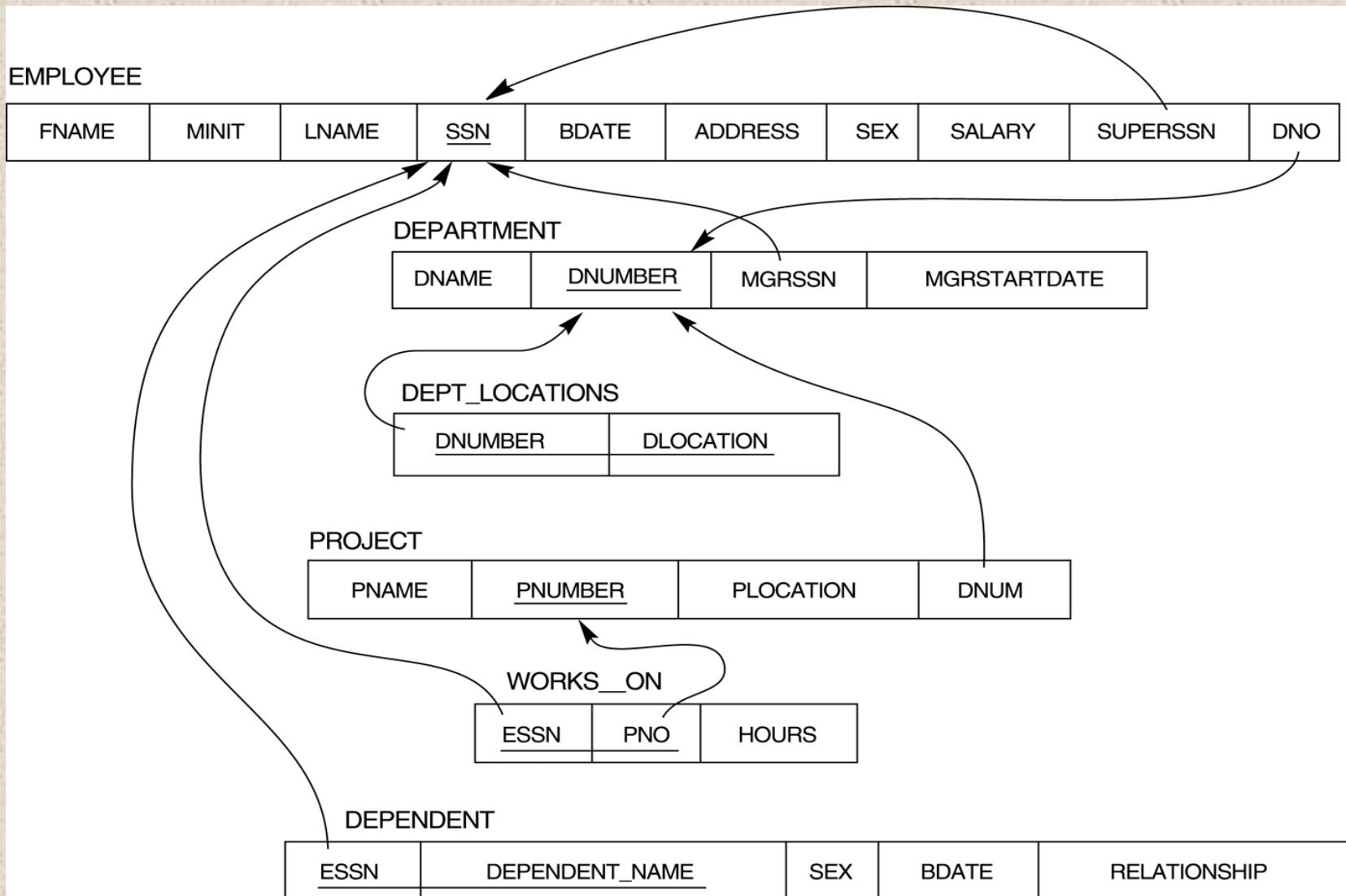


Fig: Result of mapping the COMPANY ER schema into a relational schema.

Mapping of N-ary relationship

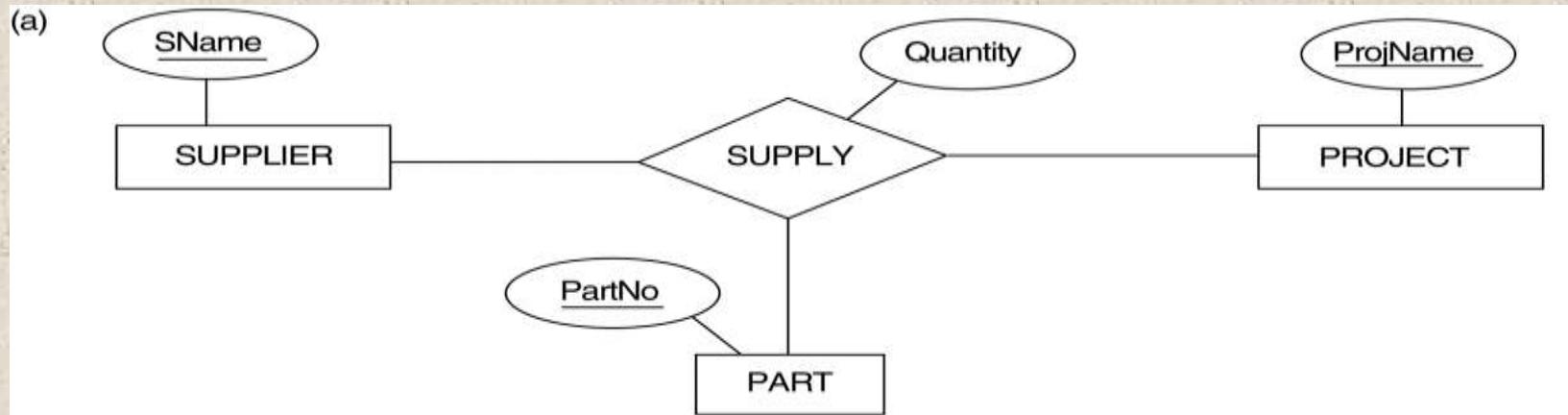


Fig: A Ternary relationship

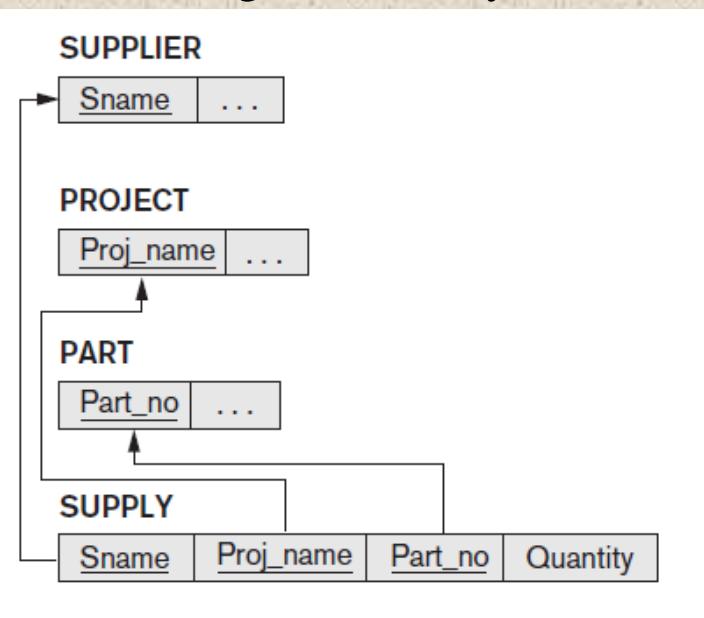


Fig: Result of ER to Relational Mapping of the n-ary relationship type

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	<i>Entity</i> relation
1:1 or 1:N relationship type	Foreign key (or <i>relationship</i> relation)
M:N relationship type	<i>Relationship</i> relation and two foreign keys
<i>n</i> -ary relationship type	<i>Relationship</i> relation and <i>n</i> foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

Database Management System (MDS 505)

Jagdish Bhatta

Unit-2

The Relational Languages and Relational Model: Relational Algebra and Relational Calculus

Relational Algebra and Calculus

- ◆ **The relational algebra** is a procedural query language. A relational algebra query describes a procedure for computing the output relation from the input relations by applying the relational algebra operators.
- ◆ The relational algebra defines a set of operators from set theory as *Union*, *Intersection*, *Set Difference*, *Cartesian Product*. Similarly, other operations as *Select*, *Project & Join* are incorporated with in the algebra operations. These operations can be categorized as unary or binary according as the number of relation on which they operate on.
- ◆ **Importance of Relational Algebra:**

The relational algebra is very important for several reasons:

- *First*, it provides a formal foundation for relational model operations.
- *Second* and perhaps more important, it is used as a basis for implementing and optimizing queries in relational DBMS (RDBMS).
- *Third*, some of its concepts are incorporated into the SQL standard query language for RDBMS

Relational Algebra and Calculus

- ◆ Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in a variation of relational calculus known as the tuple relational calculus.

Unary Relational Operations

- ◆ **SELECT Operation :-** SELECT operation is used to select **a subset of the tuples** from a relation that satisfy a *selection condition*.
- ◆ In general, the select operation is denoted by the symbol σ (sigma) as: $\sigma_{<\text{selection condition}>}(\mathbf{R})$.

The selection condition is a Boolean expression specified on the attributes of relation R and composed of *attributes name* <comparison operator> *attribute name* or *constant value*. {Comparison operator: $<$, $>$, $=$, \neq , \leq , \geq }

- ◆ Consider an example:

- ◆ To select those tuples of Employee having salary greater than 20000, following notation is used:

$\sigma_{\text{Salary} > 20,000} (\text{EMPLOYEE})$

- ◆ To select those tuples of Employee having salary equal to 10000 and who lives in Alaska, following notation is used:

$\sigma_{\text{Salary} = 10,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE})$

Jagdish Bhatta

Employee

Eid	Name	Salary	Address	Dno
011	Adam	10000	Alaska	4
022	Hilton	30000	Nevada	2

Unary Relational Operations

- ◆ **SELECT Operation :-**
- ◆ Properties of SELECT operation:
 - The SELECT operation $\sigma_{<\text{selection condition}>} (R)$ produces a relation S that has the same schema as R
 - The SELECT operation σ is ***commutative***; i.e. $\sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>} (R)) = \sigma_{<\text{condition2}>} (\sigma_{<\text{condition1}>} (R))$
 - A cascaded SELECT operation ***may be applied in any order***; i.e.
$$\sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>} (\sigma_{<\text{condition3}>} (R))) = \sigma_{<\text{condition2}>} (\sigma_{<\text{condition3}>} (\sigma_{<\text{condition1}>} (R)))$$
 - A cascaded SELECT operation may be replaced by a single selection with a conjunction of all the conditions; i.e.
$$\sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>} (\sigma_{<\text{condition3}>} (R))) = \sigma_{<\text{condition1}>} \text{ AND } <\text{condition2}> \text{ AND } <\text{condition3}> (R))$$
 - **$\sigma_{\text{Salary} = 10,000 \text{ AND Address} = \text{"Alaska"} (\text{EMPLOYEE})}$ Or ,**
 - **$\sigma_{\text{Salary} = 10,000} (\sigma_{\text{Address} = \text{"Alaska"} (\text{EMPLOYEE})})$ Or Equivalently,**

Properties of SELECT operation:.....

- The *SELECT* operation is ***unary***. It operates on only one relation.
- Selection conditions are applied to each tuple *individually* in the relation. Hence the condition cannot be span more than one tuple.
- The degree of the relation resulting from $\sigma_{\mathbf{C}}(R)$ is the same as that of R i.e. $|\sigma_{\mathbf{C}}(R)| \leq |R|$

Unary Relational Operations

- ◆ **PROJECT Operation :-** This operation selects certain *columns* from the table and discards the other columns. The PROJECT creates a vertical partitioning – one with the needed columns (attributes) containing results of the operation and other containing the discarded Columns. The general form of the project operation is:

$$\pi_{\langle \text{attribute list} \rangle}(\mathbf{R})$$

- ◆ **Consider an example:**

- ◆ To list each employee's name & salary, we can use the PROJECT operation as;

$$\pi_{\text{Name, Salary}}(\text{EMPLOYEE})$$

Employee

Eid	Name	Salary	Address	Dno
011	Adam	20000	Alaska	4
022	Hilton	30000	Nevada	2

- ◆ To retrieve Eid and Name of Employees having salary equal to 20000 and who lives in Alaska, following notation is used:

$$\pi_{\text{Eid, Name}} (\sigma_{\text{Salary} = 20,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE}))$$

Or,

$$\text{temp} = \sigma_{\text{Salary} = 20,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE})$$

$$\text{result} = \pi_{\text{Eid, Name}} (\text{temp})$$

Jagdish Bhatta

Unary Relational Operations

- ◆ Consider an example:
- ◆ To list each employee's name & salary, we can use the PROJECT operation as;

$\pi_{\text{Name, Salary}}(\text{EMPLOYEE})$

Name	Salary
Adam	20000
Hilton	30000

Employee

Eid	Name	Salary	Address	Dno
011	Adam	20000	Alaska	4
022	Hilton	30000	Nevada	2

- ◆ To retrieve Eid and Name of Employees having salary equal to 20000 and who lives in Alaska, following notation is used:

$\pi_{\text{Eid, Name}} (\sigma_{\text{Salary} = 20,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE}))$

Or,

$\text{TEMP} = \sigma_{\text{Salary} = 20,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE})$

$\text{RESULT} = \pi_{\text{Eid, Name}} (\text{TEMP})$ Jagdish Bhatta

Unary Relational Operations

Consider an example:

- ◆ To list each employee's name & salary, we can use the PROJECT operation as;

$\pi_{\text{Name, Salary}}(\text{EMPLOYEE})$

- ◆ To retrieve Eid and Name of Employees having salary equal to 20000 and who lives in Alaska, following notation is used:

$\pi_{\text{Eid, Name}} (\sigma_{\text{Salary} = 20,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE}))$

Or,

$\text{TEMP} = \sigma_{\text{Salary} = 20,000 \text{ AND Address} = \text{"Alaska}} (\text{EMPLOYEE})$

$\text{RESULT} = \pi_{\text{Eid, Name}} (\text{temp})$

EMPLOYEE

Eid	Name	Salary	Address	Dno
011	Adam	20000	Alaska	4
022	Hilton	30000	Nevada	2

RESULT

Eid	Name
011	Adam

TEMP

Eid	Name	Salary	Address	Dno
011	Adam	20000	Alaska	4

Unary Relational Operations

- ◆ **PROJECT Operation :-**
- ◆ Properties of PROJECT operation:
 - The number of tuples in the result of projection $\pi_{<\text{list}>} (R)$ is always less or equal to the number of tuples in R.
 - The *PROJECT* operation is *unary*. It operates on only one relation.
- ◆ The PROJECT operation removes any duplicate tuples. So the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**.
 - *PROJECT* is not commutative. i.e. $\pi_{<\text{list1}>} (\pi_{<\text{list2}>} (R)) = \pi_{<\text{list1}>} (R)$ as long as $<\text{list2}>$ contains the attributes in $<\text{list1}>$. Else the operation is an incorrect expression.

Unary Relational Operations

- ◆ **RENAME Operation :-** The rename operation can rename either the relation name or the attribute name or both. ρ is used to represent the rename operation. The general Rename operation can be expressed by any of the following forms:
 - $\rho_{S(B_1, B_2, \dots, B_n)}(R)$ is a renamed relation S based on R with column names B_1, B_2, \dots, B_n .
 - $\rho_S(R)$ is a renamed relation S based on R (which does not specify column names).
 - $\rho_{(B_1, B_2, \dots, B_n)}(R)$ is a renamed relation with column names B_1, B_2, \dots, B_n which does not specify a new relation name.
- ◆ **Consider an example:**
- ◆ Retrieve name and salary of employee who work in department 2.

$$\rho_{\text{RESULT}}(\pi_{\text{Name, Salary}}((\sigma_{\text{DNO} = 2}(\text{EMPLOYEE})))$$

Unary Relational Operations

- ◆ **RENAME Operation :-**
- ◆ Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, and using the **assignment operation**, denoted by \leftarrow (left arrow). It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.
- ◆ Retrieve name and salary of employee who work in department 2.

$\text{TEMP} \leftarrow \sigma_{\text{DNO}=2}(\text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{Name}, \text{Salary}}(\text{TEMP})$

Unary Relational Operations

- ◆ Consider an example:
- ◆ Retrieve name and salary of employee who work in department 2.

$$\rho_{\text{RESULT}}(\text{Emp_name}, \text{Emp_salary}) (\pi_{\text{Name}, \text{Salary}}(\sigma_{\text{DNO} = 2}(\text{EMPLOYEE})))$$

EMPLOYEE

Eid	Name	Salary	Address	Dno
011	Adam	20000	Alaska	4
022	Hilton	30000	Nevada	2
033	Ram	25000	Ktm	2

RESULT

Emp_name	Emp_salary
Hilton	30000
Ram	25000

$$\rho_{\text{RESULT1}} (\pi_{\text{Name}, \text{Salary}}(\sigma_{\text{DNO} = 2}(\text{EMPLOYEE})))$$

RESULT1

Name	Salary
Hilton	30000
Ram	25000

Unary Relational Operations

- ◆ Consider an example:
- ◆ Retrieve name and salary of employee who work in department 2.

$\rho_{\text{RESULT}} (\pi_{\text{Name, Salary}}((\sigma_{\text{DNO} = 2}(\text{EMPLOYEE}))))$

Or, $\text{TEMP} \leftarrow \sigma_{\text{DNO}=2}(\text{EMPLOYEE})$;

$\text{RESULT} \leftarrow \pi_{\text{Name, Salary}}(\text{TEMP})$

Return employees working in department no 2 ----- $\text{TEMP} \leftarrow \sigma_{\text{DNO} = 2}(\text{EMPLOYEE})$

Eid	Name	Salary	Address	Dno
022	Hilton	30000	Nevada	2
033	Ram	25000	Ktm	2

TEMP

Employee

Eid	Name	Salary	Address	Dno
011	Adam	20000	Alaska	4
022	Hilton	30000	Nevada	2
033	Ram	25000	Ktm	2

Return name and salary employees working in department no 2

$\text{Result} \leftarrow \pi_{\text{Name, Salary}}(\text{Temp})$

Name	Salary
Hilton	30000
Ram	25000

RESULT

Return Eid and Address employees working in department no 2

$\text{Result1} \leftarrow \pi_{\text{Eid, Address}}(\text{Temp})$

Eid	Address
022	Nevada
033	Ktm

Relational Algebra Operations from Set Theory

- ◆ The UNION, INTERSECTION, and SET DIFFERENCE operation:
 - These operations are binary operations.
 - The two relations on which any of the three operations are applied must have the same type of tuples; this condition is called ***union compatibility***. Two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ are said to be ***union or type compatible*** if they have the same degree n and if $\text{dom}(A_i) = \text{dom}(B_i)$, for $1 \leq i \leq n$. This means that the two relations have the same no. of attributes, and each corresponding pair of attributes has the same domain.
(Field names are not used in defining union compatibility)

Operations from Set Theory

- ◆ **UNION Operation :-** For any two union-compatible relations R and S, the union operation denoted by $R \cup S$ is a relation that includes all the tuples that are either in R or in S or in both R and S. Here duplicate tuples are eliminated, if any. So it can be defined as:

$$R \cup S = \{t \mid t \in R \text{ or } t \in S\}$$

- ◆ For Example:

Student

Fname	Lname
Bryan	Adam
Michal	Clark
Susan	Yao

Teacher

Fname	Lname
Ramez	Elmsari
Michal	Clark
Jimmy	Wang

Student \cup Teacher

Fname	Lname
Bryan	Adam
Michal	Clark
Susan	Yao
Ramez	Elmsari
Jimmy	Wang

Operations from Set Theory

- ◆ **INTERSECTION Operation :-** For any two union-compatible relations R and S, the intersection operation denoted by $R \cap S$ is a relation that includes all the tuples that are in both R and S. So it can be defined as:

$$R \cap S = \{t \mid t \in R \text{ and } t \in S\}$$

- ◆ For Example:

Student

Fname	Lname
Bryan	Adam
Michal	Clark
Susan	Yao

Teacher

Fname	Lname
Ramez	Elmsari
Michal	Clark
Jimmy	Wang

Student \cap Teacher

Fname	Lname
Michal	Clark

Operations from Set Theory

- ◆ **SET DIFFERENCE (Minus) Operation :-** For any two union-compatible relations R and S, the set difference operation denoted by R - S is a relation that includes all the tuples that are in R but not in S. So it can be defined as:

$$R - S = \{t \mid t \in R \text{ and } t \notin S\}$$

- ◆ For Example:

Student

Fname	Lname
Bryan	Adam
Michal	Clark
Susan	Yao

Teacher

Fname	Lname
Ramez	Elmsari
Michal	Clark
Jimmy	Wang

Student - Teacher

Fname	Lname
Bryan	Adam
Susan	Yao

Operations from Set Theory

◆ Properties of Union, Intersection, Set Difference:-

- Both UNION & INTERSECTION are commutative while SET DIFFERENCE is not.

i.e. $R \cup S = S \cup R$, and $R \cap S = S \cap R$

But, $R - S \neq S - R$

- Both UNION and INTERSECTION are associative.

i.e. $R \cup (S \cup T) = (R \cup S) \cup T$

And, $(R \cap S) \cap T = R \cap (S \cap T)$

Note:

For any two type compatible relations, R_1 & R_2 . The resulting relation for $R_1 \cup R_2$, $R_1 \cap R_2$, or $R_1 - R_2$ has the same attribute names as the first operand relation R_1 (by convention).

Operations from Set Theory

Note:

For any two type compatible relations, R_1 & R_2 . The resulting relation for $R_1 \cup R_2$, $R_1 \cap R_2$, or $R_1 - R_2$ has the same attribute names as the first operand relation R_1 (by convention).

Student

Fname	Lname
Bryan	Adam
Michal	Clark
Susan	Yao

Teacher

Firstname	Lastname
Ramez	Elmsari
Michal	Clark
Jimmy	Wang

Student \cap Teacher

Fname	Lname
Michal	Clark

Operations from Set Theory

Note:

For any two type compatible relations, R_1 & R_2 . The resulting relation for $R_1 \cup R_2$, $R_1 \cap R_2$, or $R_1 - R_2$ has the same attribute names as the first operand relation R_1 (by convention).

Student

Fname	Lname
Bryan	Adam
Michal	Clark
Susan	Yao

Teacher

Firstname	Lastname
Ramez	Elmsari
Michal	Clark
Jimmy	Wang

Teacher \cap Student

Firstname	Lastname
Michal	Clark

Operations from Set Theory

Student

Fname	Lname	Adress
Ramez	Elmsari	Ktm

Teacher

Firstname	Lastname
Ramez	Elmsari
Michal	Clark
Jimmy	Wang

Manager

Fname	Mid
Ramez	011

Here $\text{Teacher} \{\cap, \cup, -\} \text{Student}$ is not possible. Since relations student and teacher are not union compatible. Since number of attributes are not same.

Here $\text{Teacher} \{\cap, \cup, -\} \text{Manager}$ is not possible. Since relations manager and teacher are not union compatible. Since domain of corresponding attributes is not same i.e. domain of Lastname is string while domain of Mid is integer.

Binary Relational Operations

◆ **Cartesian Product Operation :-**

- Often known as cross join or Cartesian join.
- The relations on which it is to be applied do not have to be union compatible.
- This operation is used to combine tuples from two relations in combinatorial fashion.
- It is denoted by \times .
- For any two relations $R(A_1, A_2, \dots, A_n)$ & $S(B_1, B_2, \dots, B_m)$ the cross product $R \times S$ is a relation Q with degree $n + m$, $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ in the same order as the attributes appear in R & S . The resulting relation Q has one tuple for each combination of tuples—one from R and one from S . So here we will have $n \times m$ tuples.

Binary Relational Operations

◆ Cartesian Product Operation :-

– Consider an Example:

Employee

<u>SSN</u>	Fname	Lname	Dno
111	Bryan	Adams	1
222	John	Smith	2
333	Ravi	Sastri	2

Department

<u>Dno</u>	Dname	MgrSSN
1	HRM	111
2	Account	222

Employee × Department

Employee.SSN	Employee.Fname	Employee.Lname	Employee.Dno	Department.Dno	Department.Dname	Department.MgrSSN
111	Bryan	Adams	1	1	HRM	111
111	Bryan	Adams	1	2	Account	222
222	John	Smith	2	1	HRM	111
222	John	Smith	2	2	Account	222
333	Ravi	Sastri	2	1	HRM	111
333	Ravi	Sastri	2	2	Account	222

Binary Relational Operations

◆ JOIN Operation :-

- This operation is used to combine related tuples from two relations into single tuples. It is very useful for any relational DB with more than a single relation because it allows us to process relationships among relations. JOIN operation is denoted by \bowtie .
- The general form of a join operation on two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is $R \bowtie_{\text{join condition}} S$.
- Each tuple in $R \bowtie S$ is a combination of one tuple from R & one from S whenever the combination satisfies the *join condition*. The join condition is specified on attributes from two relations R & S and is evaluated for each combination of tuples.
- Thus in join, only the condition satisfying tuples appear in result, whereas in Cartesian product all the combination of tuples are included in the result.

Binary Relational Operations

◆ Join Operation :-

- Consider an Example:

Employee

SSN	Fname	Lname	Dno
111	Bryan	Adams	1
222	John	Smith	2
333	Ravi	Sastri	2

Department

Dno	Dname	MgrSSN
1	HRM	111
2	Account	222

Employee \bowtie Employee.Dno=Department.Dno Department

Employee.SSN	Employee.Fname	Employee.Lname	Employee.Dno	Department.Dno	Department.Dname	Department.MgrSSN
111	Bryan	Adams	1	1	HRM	111
222	John	Smith	2	2	Account	222
333	Ravi	Sastri	2	2	Account	222

Binary Relational Operations

◆ JOIN Operation :-

◆ Example:

- To select the name of Managers of each departments. The relational algebra query can be written as;

$$\text{Mgrlist} \leftarrow \text{Employee} \bowtie_{\text{SSN} = \text{MgrSSN}} \text{Department}$$
$$\text{Result} \leftarrow \pi_{\text{FName}, \text{LName}}(\text{Mgrlist})$$

- To find the name of Managers having last name “Adams”. The relational algebra query can be written as;

$$\text{Mgrlist} \leftarrow \text{Employee} \bowtie_{\text{SSN} = \text{MgrSSN}} \text{Department}$$
$$\text{Temp} \leftarrow \sigma_{\text{LName} = \text{“Adams”}} (\text{Mgrlist})$$
$$\text{Result} \leftarrow \pi_{\text{FName}, \text{LName}}(\text{Temp})$$

Binary Relational Operations

- ◆ **Join Operation :-** To select the name of Managers of each departments. The relational algebra query can be written as;

Employee

SSN	Fname	Lname	Dno
111	Bryan	Adams	1
222	John	Smith	2
333	Ravi	Sastri	2

Department

Dno	Dname	MgrSSN
1	HRM	111
2	Account	222

Mgrlist \leftarrow Employee $\bowtie_{SSN=MgrSSN}$ Department

Employee.SSN	Employee.Fname	Employee.Lname	Employee.Dno	Department.Dno	Department.Dname	Department.MgrSSN
111	Bryan	Adams	1	1	HRM	111
222	John	Smith	2	2	Account	222

Result $\leftarrow \pi_{FName, LName}(Mgrlist)$

Employee. Fname	Employee.Lname
Bryan	Adams
John	Smith

Equivalently:

$\pi_{FName, LName}(Employee \bowtie_{SSN=MgrSSN} Department)$

Binary Relational Operations

- ◆ **Join Operation :-** To find the name of Managers having last name “Adams”.

The relational algebra query can be written as

$$Mgelist \leftarrow Employee \bowtie_{SSN=MgrSSN} Department$$

Employee.SSN	Employee.Fname	Employee.Lname	Employee.Dno	Department.Dno	Department.Dname	Department
111	Bryan	Adams	1	1	HRM	111
222	John	Smith	2	2	Account	222

$$Temp \leftarrow \sigma_{LName="Adams"}(Mgelist)$$

Employee.SSN	Employee.Fname	Employee.Lname	Employee.Dno	Department.Dno	Department.Dname	Department
111	Bryan	Adams	1	1	HRM	111

$$Result \leftarrow \pi_{FName, LName}(Temp)$$

Employee. Fname	Employee.Lname
Bryan	Adams

Equivalently:

$$\pi_{FName, LName}(\sigma_{LName="Adams"}(Employee \bowtie_{SSN=MgrSSN} Department))$$

Jagdish Bhatta

Binary Relational Operations

- ◆ **Theta JOIN Operation :-**
- ◆ Join operation between relations R and S with join condition $A_i \theta B_i$;
 A_i and B_i are attributers belonging to relations R and S respectively
and have the same domain. And $\theta = \{ =, <, >, \leq, \geq, \neq \}$.
- ◆ Thus, join operation with generalized join condition is the **theta join** and has following notation;

$$R \bowtie_{A_i \theta B_i} S$$

Binary Relational Operations

- ◆ **EQUIJOIN Operation :-**
- ◆ The join operation where the join condition involves only equality comparison is called equijoin. Here only the comparison operator used is “=”.
- ◆ In the result of an EQUIJOIN, we always have one or more pairs of attributes (whose names need not be identical) that have *identical values* in every tuple.
- ◆ The EQUIJOIN has following representation

$R \bowtie_{A_i=B_i} S$; where A_i and B_i are attributes of relations R & S respectively.

Binary Relational Operations

- ◆ **Natural Join Operation :-**
- ◆ In equijoin operation, the resulting relation have a pair or more attributes with identical value so having multiple fields in a tuple with same value is unnecessary. To get rid of it, a new join type called **natural join** is introduced. This natural join is denoted by * or \bowtie .
- ◆ The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, have the same name in both relations. If this is not the case, a renaming operation is applied first.
- ◆ Example:
 - To get records of Managers of each departments, the relational algebra query can be written as;

Mgrlist \leftarrow Employee * $\rho_{(Dno, Dname, SSN)} Department$

In this realtion Mgrlist, it will have just single entry SSN rather than having both SSN & MgrSSN as in equijoin, since MgrSSN in Department is renamed to SSN. Hence natural join will result.

- ◆ In general, natural join is performed by equating all attribute pairs that have the same name in two relations.

Binary Relational Operations

- ◆ **Natural Join Operation :-** Select name of employees and their departments.

Employee

SSN	Fname	Lname	Dno
111	Bryan	Adams	1
222	John	Smith	2
333	Ravi	Sastri	2

Department

Dno	Dname	MgrSSN
1	HRM	111
2	Account	222

Mgrlist \leftarrow Employee * Department

SSN	Fname	Lname	Dno	Dname	MgrSSN
111	Bryan	Adams	1	HRM	111
222	John	Smith	2	Account	222
333	Ravi	Sastri	2	Account	222

Result $\leftarrow \pi_{\text{FName}, \text{LName}, \text{Dname}} (\text{Mgrlist})$

Fname	Lname	Dname
Bryan	Adams	HRM
John	Smith	Account
Ravi	Sastri	Account

Equivalently:

$\pi_{\text{FName}, \text{LName}, \text{Dname}} (\text{Employee} * \text{Department})$

Consider few Examples:

- ◆ Suppose we have following relations;

Employee (Fname, Minit, Lname, SSN, Bdate, Address, Sex, Salary, SuperSSN, Dno)

Department (Dname, Dnumber, MgrSSN, Mgrsstartdate)

Project (Pnumber, Pname, Plocation, Dnum)

Works_on (ESSN, Pno, Hours)

Dep_Locations (Dnumber, Dlocation)

Dependent (ESSN, Dep_name, Sex, Bdate, Relationship)

Consider few Examples:

- ◆ Select employees having salary greater than 30000
- ◆ Select project name and project location of projects
- ◆ Select all employees having salary less than 1000 and address “KTM”
- ◆ Select project name and project location of projects having project name “Construction”

Consider few Examples:

- ◆ Select employees having salary greater than 30000

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

- ◆ Select project name and project location of projects

$$\pi_{\text{Pname}, \text{Plocation}}(\text{Project})$$

- ◆ Select all employees having salary less than 1000 and address “KTM”

$$\sigma_{\text{Salary} < 30000 \text{ AND } \text{Address} = \text{“KTM”}}(\text{Employee})$$

- ◆ Select project name and project location of projects having project name “Construction”

$$\pi_{\text{Pname}, \text{Plocation}}(\sigma_{\text{Pname} = \text{“Construction”}}(\text{Project}))$$

- ♦ Lets try few more queries:
 - Retrieve the name and address of all employee who works for the “Research Department”.
 - Retrieve the list of project numbers that “John Smith” works on.

- ♦ Lets try few more queries:
 - Retrieve the name and address of all employee who works for the “Research Department”.

$\text{Research_dept} \leftarrow \sigma_{\text{DName}=\text{"Research"}}(\text{Department})$

$\text{REmp} \leftarrow \text{Research_dept} \bowtie_{\text{Dnumber}=\text{Dno}}(\text{Employee})$

$\text{Result} \leftarrow \pi_{\text{FName LName, Address}}(\text{REmp})$

Or,

$\text{REmp} \leftarrow \text{Department} \bowtie_{\text{Dnumber}=\text{Dno}}(\text{Employee})$

$\text{Research_dept} \leftarrow \sigma_{\text{DName}=\text{"Research"}}(\text{REmp})$

$\text{Result} \leftarrow \pi_{\text{FName LName, Address}}(\text{Research_dept})$

Or,

$\pi_{\text{FName LName, Address}}(\sigma_{\text{DName}=\text{"Research"}}(\text{Department} \bowtie_{\text{Dnumber}=\text{Dno}}(\text{Employee})))$

- Retrieve the list of project numbers that “John Smith” works on.

$\text{Emp_John} \leftarrow \sigma_{\text{FName}=\text{"John"} \text{ AND } \text{LName}=\text{"Smith"}}(\text{Employee})$

$\text{John_Proj} \leftarrow \text{Emp_John} \bowtie_{\text{SSN}=\text{ESSN}}(\text{Works_on})$

$\text{Result} \leftarrow \pi_{\text{Pno}}(\text{John_Proj})$

Or,

$\text{Emp_John} \leftarrow \pi_{\text{SSN}}(\sigma_{\text{FName}=\text{"John"} \text{ AND } \text{LName}=\text{"Smith"}}(\text{Employee}))$

$\text{John_Proj} \leftarrow (\rho_{(\text{ESSN})} \text{Emp_John}) * (\text{Works_on})$

$\text{Result} \leftarrow \pi_{\text{Pno}}(\text{John_Proj})$

- Retrieve the name of employee who have no dependents.
- Retrieve the list names of employees and projects they are working on

- Retrieve the name of employee who have no dependents.

$\text{Emp_List} \leftarrow \pi_{\text{SSN}}(\text{Employee})$

$\text{Emp_Depend} \leftarrow \pi_{\text{ESSN}}(\text{Dependent})$

$\text{Emp_Nodepend} \leftarrow \text{Emp_List} - \text{Emp_Depend}$

$\text{Emp} \leftarrow \text{Employee} * \text{Emp_Nodepend}$

$\text{Result} \leftarrow \pi_{\text{FName, Lname}} (\text{Emp})$

- Retrieve the list names of employees and projects they are working on

$\text{Working_Proj} \leftarrow \text{Project} \bowtie_{\text{Pnumber}=\text{Pno}} (\text{Works_on})$

$\text{Employee_Proj} \leftarrow \text{Working_Proj} \bowtie_{\text{ESSN}=\text{SSN}} (\text{Employee})$

$\text{Result} \leftarrow \pi_{\text{FName, Lname, Pname}} (\text{Employee_Proj})$

Complete Set of Relational Operations

- ◆ The set of operations including *select* σ , *project* π , *union* \cup , *set difference* - , *and Cartesian product* X is called a complete set because any other relational algebra expression can be expressed by a combination of these five operations.

Binary Relational Operations

- ◆ **Division Operation :-**
- ◆ Let r and s be relations on schemas R and S respectively where

$$R = (A_1, \dots, A_m, B_1, \dots, B_n) \text{ & } S = (B_1, \dots, B_n)$$

The result of $r \div s$ is a relation on schema

$$R \div S = (A_1, \dots, A_m)$$

- ◆ Alternatively to define the operation, consider two relations A & B in which A has two fields x & y and B has just one field y with the same domain as in A , then we define $A \div B$ as a set of all x values in unary tuple such that for every y value in B , there is a tuple (x, y) in A .

- ◆ **Eg:-**

A	X	Y	B	Y	$A \div B$	X
	$x1$	$y1$		$y1$		$x1$
	$x2$	$y1$		$y2$		$x2$
	$x1$	$y2$				
	$x2$	$y2$				
	$x3$	$y1$				
	$x4$	$y2$				

Binary Relational Operations

- ◆ **Division Operation :-**
- ◆ Generally, Division operation is suited to queries that include the phrase “for all”.
- ◆ **Eg: Retrieve the names of employees who work on all the projects that ‘John Smith’ works on.**
- ◆ First retrieve the list of project numbers that ‘John Smith’ works on in the intermediate relation SMITH_PNOS:

$$\text{SMITH} \leftarrow \sigma_{\text{Fname}=\text{'John'} \text{ AND } \text{Lname}=\text{'Smith'}}(\text{EMPLOYEE})$$
$$\text{SMITH_PNOS} \leftarrow \pi_{\text{Pno}}(\text{WORKS_ON} \text{ JOIN } \text{Essn}=\text{Ssn} \text{ SMITH})$$

- ◆ Next, create a relation that includes a tuple $\langle \text{Pno}, \text{Essn} \rangle$ 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:

$$\text{SSNS(Ssn)} \leftarrow \text{SSN_PNOS} \div \text{SMITH_PNOS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname}}(\text{SSNS} * \text{EMPLOYEE})$$

Binary Relational Operations

◆ Division Operation :-

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

(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

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

$$\text{DEPT5_PROJS} \leftarrow \rho_{(\text{Pno})}(\pi_{\text{Pnumber}}(\sigma_{\text{Dnum}=5}(\text{PROJECT})))$$

$$\text{EMP_PROJ} \leftarrow \rho_{(\text{Ssn}, \text{Pno})}(\pi_{\text{Essh}, \text{Pno}}(\text{WORKS_ON}))$$

$$\text{RESULT_EMP_SSNS} \leftarrow \text{EMP_PROJ} \div \text{DEPT5_PROJS}$$

$$\text{RESULT} \leftarrow \pi_{\text{Lname}, \text{Fname}}(\text{RESULT_EMP_SSNS} * \text{EMPLOYEE})$$

- 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. Notice that we renamed the attributes so that they will be correctly used in 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.

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:

$$\pi_{F_1, F_2, \dots, F_n}(R)$$

- ◆ F_1, F_2, \dots, F_n 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.

Additional Relational Operations

- ◆ **Generalized Projection:-**

- ◆ 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:

$\text{REPORT} \leftarrow \rho_{(\text{Ssn}, \text{Net_salary}, \text{Bonus}, \text{Tax})}(\pi_{\text{Ssn}, \text{Salary} - \text{Deduction}, 2000 * \text{Years_service}, 0.25 * \text{Salary}}(\text{EMPLOYEE}))$

Additional Relational Operations

◆ Aggregate Functions and Groupings:-

- SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT
- Aggregate functions take a collection of values and return a single value as a result.
- Aggregate function operations are defined using the symbol \mathcal{F} (pronounced “script F”), or \mathcal{g} (pronounced “calligraphic G”).
- The general form of *aggregate operation* in relational algebra

$G_1, G_2, \dots, G_n \mathcal{F} F_1(A_1), F_2(A_2), \dots, F_n(A_n) (E)$,

where, E is any relational-algebra expression

G_1, G_2, \dots, G_n is a list of attributes on which to group (can be empty)

Each F_i is an aggregate function

Each A_i is an attribute name

Additional Relational Operations

◆ Aggregate Functions and Groupings:-

- Consider an example;

R			$\mathcal{F}_{\text{sum}(c)}(R)$
A	B	C	Sum_C
A1	B1	7	
A2	B2	3	
A3	B3	10	20

- Now using aggregate function, the query to find the total number of employees and their average salary will be:

$\mathcal{F}_{\text{COUNT(SSN), AVERAGE(Salary)}}(\text{Employee})$

The result of this query will be such as;

COUNT_SSN	AVERAGE_Salary
8	257876

Additional Relational Operations

◆ Aggregate Functions and Groupings:-

- We may group the tuples in the relation by the value of some attributes and apply aggregate function. For example:

To retrieve Department number, total number of employees and their average salary, the query may be as

Dno \mathcal{F} COUNT(SSN), AVERAGE(Salary) (Employee)

The result of this query will be such as;

Dno	COUNT_SSN	AVERAGE_Salary
1	8	257876
2	10	300000

- There are cases where we must eliminate multiple occurrences of a value before computing an aggregate function. For this, we use the function name with the addition of the hyphenated string “**distinct**” appended to the end of the function name (for example, **count-distinct**).

Additional Relational Operations

- ◆ **Outer Join:-**
- ◆ In NATURAL JOIN, tuples without a *matching* (or *related*) tuple are eliminated from the join result. Tuples with null in the join attributes are also eliminated. This amounts to loss of information. A set of operations, called *outer joins*, can be used when we want to keep all the tuples in R, or all those in S, or all those in both relations in the result of the join, regardless of whether or not they have matching tuples in the other relation.
- ◆ The *left outer join* operation keeps every tuple in the *first* or *left* relation R in $R \bowtie S$; if no matching tuple is found in S, then the attributes of S in the join result are filled or “padded” with null values. A similar operation, *right outer join*, keeps every tuple in the *second* or *right* relation S in the result of $R \bowtie S$; if no matching tuple is found in R, then the attributes of R in the join result are filled or “padded” with null values. While, the *full outer join*, denoted by $R \bowtie S$ keeps all tuples in both the left and the right relations in the result; when no matching tuples are found, they are padded with null values as needed.

Additional Relational Operations

- ◆ Outer Join:-
- ◆ Example

Employee				Department	
Eid	Ename	Address	Dno	Dno	Dname
1	Ram	KTM	111		
2	Rita	PKR	222		
3	Hari	KTM	333		

Employee $\bowtie \times$ Department

Eid	Ename	Address	Dno	Dname
1	Ram	KTM	111	HRM
2	Rita	PKR	222	Admin
3	Hari	KTM	333	NULL

Employee $\bowtie \square$ Department

Eid	Ename	Address	Dno	Dname
1	Ram	KTM	111	HRM
2	Rita	PKR	222	Admin
NULL	NULL	NULL	444	Account

Employee $\bowtie \bowtie \square$ Department

Eid	Ename	Address	Dno	Dname
1	Ram	KTM	111	HRM
2	Rita	PKR	222	Admin
3	Hari	KTM	333	NULL
NULL	NULL	NULL	444	Account

Tuple Relational Calculus

- ◆ In tuple relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how, or *in what order*, to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request *in a particular order* of applying the operations; thus, it can be considered as a **procedural** way of stating a query.

Tuple Relational Calculus

- ◆ It has been shown that any retrieval that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the languages is *identical*. This led to the definition of the concept of a *relationally complete* language. A relational query language L is considered **relationally complete** if we can express in L any query that can be expressed in relational calculus.

Tuple Relational Calculus

- ◆ **Tuple Variables and Range Relations**
- ◆ The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form:

$$\{t \mid \text{COND}(t)\}$$

- ◆ where t is a tuple variable and $\text{COND}(t)$ is a conditional (Boolean) expression involving t that evaluates to either TRUE or FALSE for different assignments of tuples to the variable t . The result of such a query is the set of all tuples t that evaluate $\text{COND}(t)$ to TRUE. These tuples are said to **satisfy** $\text{COND}(t)$.

Tuple Relational Calculus

- ◆ **Tuple Variables and Range Relations**
- ◆ For example, to find all employees whose salary is above \$50,000, we can write the following tuple calculus expression:

$$\{t \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Salary} > 50000\}$$

- ◆ The condition $\text{EMPLOYEE}(t)$ specifies that the **range relation** of tuple variable t is EMPLOYEE . Each EMPLOYEE tuple t that satisfies the condition $t.\text{Salary} > 50000$ will be retrieved. Notice that $t.\text{Salary}$ references attribute Salary of tuple variable t .
- ◆ The previous 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

$$\{t.\text{Fname}, t.\text{Lname} \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Salary} > 50000\}$$

Tuple Relational Calculus

- ◆ **Tuple Variables and Range Relations**
- ◆ Retrieve the birth date and address of the employee (or employees) whose name is John B. Smith.

Tuple Relational Calculus

- ◆ **Tuple Variables and Range Relations**
- ◆ Retrieve the birth date and address of the employee (or employees) whose name is John B. Smith.

$$\{t.Bdate, t.Address \mid \text{EMPLOYEE}(t) \text{ AND } t.Fname = \text{'John'} \text{ AND } t.Minit = \text{'B'} \text{ AND } t.Lname = \text{'Smith'}\}$$

- ◆ In tuple relational calculus, we first specify the requested attributes $t.Bdate$ and $t.Address$ for each selected tuple t . Then we specify the condition for selecting a tuple following the bar (\mid)—namely, that t be a tuple of the EMPLOYEE relation whose Fname, Minit, and Lname attribute values are ‘John’, ‘B’, and ‘Smith’, respectively.

Tuple Relational Calculus

- ◆ **Expressions and Formulas in Tuple Relational Calculus**
- ◆ A general **expression** of the tuple relational calculus is of the form
$$\{t_1.Aj, t_2.Ak, \dots, t_n.Am \mid \text{COND}(t_1, t_2, \dots, t_n, t_{n+1}, t_{n+2}, \dots, t_{n+m})\}$$
- ◆ where $t_1, t_2, \dots, t_n, t_{n+1}, \dots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula** of the tuple relational calculus.

Tuple Relational Calculus

- ◆ The Existential and Universal Quantifiers
- ◆ Two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** (\forall) and the **existential quantifier** (\exists). They are used to **bound** a tuple variable.
- ◆ Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an $(\exists t)$ or $(\forall t)$ clause; otherwise, it is free. Formally, we define a tuple variable.

Tuple Relational Calculus

- ◆ Examples (With reference to schemas in slide number 34)
- ◆ **Query 1.** List the name and address of all employees who work for the ‘Research’ department.

Q1: { t .Fname, t .Lname, t .Address | EMPLOYEE(t) AND ($\exists d$)(DEPARTMENT(d) AND d .Dname=‘Research’ AND d .Dnumber= t .Dno)}

- ◆ The *only free tuple variables* in a tuple relational calculus expression should be those that appear to the left of the bar (|). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified after the bar in Q1, the attributes Fname, Lname, and Address are retrieved for each such tuple. The conditions EMPLOYEE(t) and DEPARTMENT(d) specify the range relations for t and d . The condition d .Dname = ‘Research’ is a **selection condition** and corresponds to a SELECT operation in the relational algebra, whereas the condition d .Dnumber = t .Dno is a **join condition** and is similar in purpose to the (INNER) JOIN operation used in relational algebra query in slide number 35.

Tuple Relational Calculus

- ◆ Examples (With reference to schemas in slide number 34)
- ◆ Several tuple variables in a query can range over the same relation. For example, to specify Q2 — for each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor—we specify two tuple variables e and s that both range over the EMPLOYEE relation:

Q2: { $e.\text{Fname}, e.\text{Lname}, s.\text{Fname}, s.\text{Lname} \mid \text{EMPLOYEE}(e) \text{ AND } \text{EMPLOYEE}(s) \text{ AND } e.\text{Super_ssn}=s.\text{Ssn}$ }

Tuple Relational Calculus

- ◆ Examples (With reference to schemas in slide number 34)
- ◆ List the name of each employee who works on *some* project controlled by department number 5. In this case we need two join conditions and two existential quantifiers.
- ◆ List the names of employees who have no dependents.

Tuple Relational Calculus

- ◆ Examples (With reference to schemas in slide number 34)
- ◆ List the name of each employee who works on *some* project controlled by department number 5. In this case we need two join conditions and two existential quantifiers.

$$\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e) \text{ AND } ((\exists x)(\exists w)(\text{PROJECT}(x) \text{ AND } \text{WORKS_ON}(w) \text{ AND } x.\text{Dnum}=5 \text{ AND } w.\text{Essn}=e.\text{Ssn} \text{ AND } x.\text{Pnumber}=w.\text{Pno}))\}$$

- ◆ List the names of employees who have no dependents.

$$\{e.\text{Fname}, e.\text{Lname} \mid \text{EMPLOYEE}(e) \text{ AND } (\text{NOT } (\exists d)(\text{DEPENDENT}(d) \text{ AND } e.\text{Ssn}=d.\text{Essn}))\}$$

Equivalently, following is same;

$$\{e.\text{Fname}, e.\text{Lname} \mid \text{EMPLOYEE}(e) \text{ AND } ((\forall d)(\text{NOT}(\text{DEPENDENT}(d)) \text{ OR } \text{NOT}(e.\text{Ssn}=d.\text{Essn})))\}$$

Domain Relational Calculus

- ◆ 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

$$\{x_1, x_2, \dots, x_n \mid \text{COND}(x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m})\}$$

- ◆ where $x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}$ are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

Domain Relational Calculus

- ◆ Examples
- ◆ List the birth date and address of the employee whose name is ‘John B. Smith’.

$$\{u, v \mid (\exists q) (\exists r) (\exists s) (\exists t) (\exists w) (\exists x) (\exists y) (\exists z) (\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } q=\text{'John'} \text{ AND } r=\text{'B'} \text{ AND } s=\text{'Smith'})\}$$

- ◆ 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, \dots, 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 (\mid)—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.

Domain Relational Calculus

- ◆ Examples
- ◆ List the birth date and address of the employee whose name is ‘John B. Smith’.

$\{u, v \mid (\exists q) (\exists r) (\exists s) (\exists t) (\exists w) (\exists x) (\exists y) (\exists z) (\text{EMPLOYEE}(qrstuvwxyz)$
AND $q = \text{'John'}$ **AND** $r = \text{'B'}$ **AND** $s = \text{'Smith'})\}$

- ◆ Alternatively,

$\{u, v \mid \text{EMPLOYEE}(\text{'John'}, \text{'B'}, \text{'Smith'}, t, u, v, w, x, y, z)\}$

Domain Relational Calculus

- ◆ **Examples**
- ◆ Retrieve the name and address of all employees who work for the ‘Research’ department.

$\{q, s, v \mid (\exists z) (\exists l) (\exists m) (\text{EMPLOYEE}(qrstuvwxyz}) \textbf{ AND } \text{DEPARTMENT}(lmno) \textbf{ AND } l=\text{‘Research’} \textbf{ AND } m=z)\}$

- ◆ A condition relating two domain variables that range over attributes from two relations, such as $m = z$ in above query, is a **join condition on dno**, whereas a condition that relates a domain variable to a constant, such as $l = \text{‘Research’}$, is a **selection condition**.

Domain Relational Calculus

- ◆ Examples
- ◆ List the names of employees who have no dependents.

Domain Relational Calculus

- ◆ Examples
- ◆ List the names of employees who have no dependents.

$$\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } (\text{NOT}(\exists l)(\text{DEPENDENT}(lmnop) \text{ AND } t=l)))\}$$

Domain Relational Calculus

- ◆ Examples
- ◆ List the names of managers who have at least one dependent.

$$\{s, q \mid (\exists t)(\exists j)(\exists l)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } \text{DEPARTMENT}(hijk) \text{ AND } \text{DEPENDENT}(lmnop) \text{ AND } t=j \text{ AND } l=t)\}$$

Here, t is SSN in Employee, j is MGRSSN in Department, l is ESSN in Dependent.

Putting Altogether

◆ Examples

- ◆ Retrieve the name and address of all employees who work for the ‘Research’ department.

Employee (Fname, Minit, Lname, SSN, Bdate, Address, Sex, Salary, SuperSSN, Dno)

Department (Dname, Dnumber, MgrSSN, Mgrsstartdate)

◆ Relational Algebra

$$\pi_{\text{FName } \text{LName}, \text{Address}}(\sigma_{\text{DName}=\text{“Research”}}(\text{Department} \bowtie_{\text{Dnumber}=\text{Dno}} (\text{Employee})))$$

◆ Tuple Calculus

$$\{t.\text{Fname}, t.\text{Lname}, t.\text{Address} \mid \text{EMPLOYEE}(t) \text{ AND } (\exists d)(\text{DEPARTMENT}(d) \text{ AND } d.\text{Dname}=\text{‘Research’} \text{ AND } d.\text{Dnumber}=t.\text{Dno})\}$$

◆ Domain Calculus

$$\{q, s, v \mid (\exists z) (\exists l) (\exists m) (\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } \text{DEPARTMENT}(lmno) \text{ AND } l=\text{‘Research’} \text{ AND } m=z)\}$$

Database Management System (MDS 505)

Jagdish Bhatta

Unit-2

The Relational Languages and Relational Model: SQL

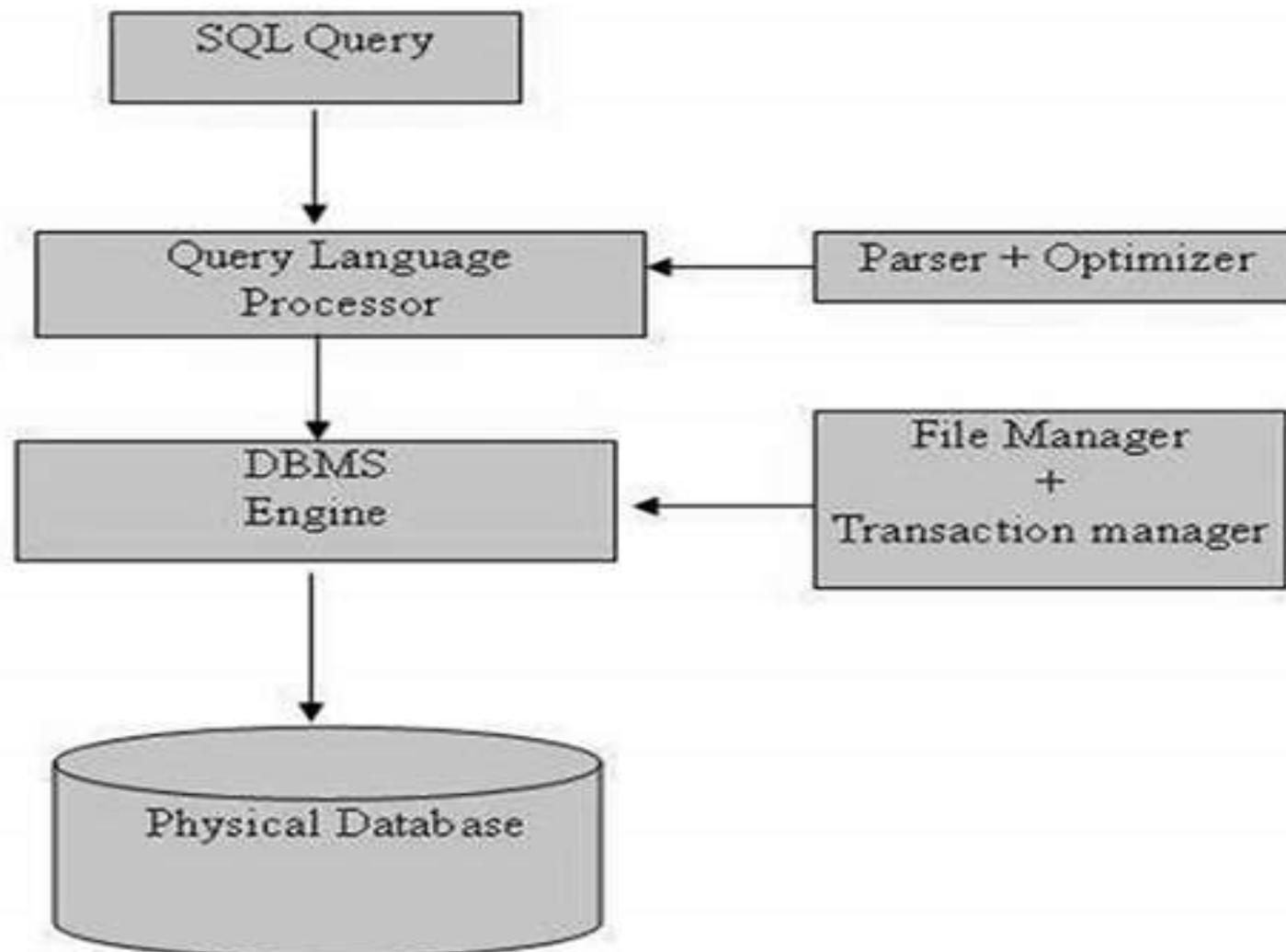
Introduction

- ◆ Originally, Structured Query Language (SQL), was called SEQUEL (for Structured English Query Language) and was designed & implemented at IBM Research.
- ◆ SQL is the most popular and most user friendly query language. SQL uses a combination of *relational-algebra* and *relational-calculus* constructs.
- ◆ Although we refer to the SQL language as a “*query language*”, it can be used for defining the structure of the data, modifying the data in the database, and specifying security constraints.
- ◆ SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Introduction

- ◆ SQL has the following features:
 - **Data-definition language(DDL):** - The SQL DDL provides commands for defining relation schemas and modifying relation schemas.
 - **Interactive data-manipulation language(DML):** - The SQL DML includes a query language based on both the relational algebra and tuple relational calculus. It also includes commands to insert, delete, and modify tuples.
 - **View definition:** - The SQL DDL includes commands for defining views.
 - **Transaction control:** - SQL includes commands for specifying the beginning and ending of transactions.
 - **Embedded SQL and dynamic SQL:** - Embedded SQL and dynamic SQL defines how SQL statements can be embedded within general purpose programming languages, such as PHP, Java etc.
 - **Integrity:** - The SQL DDL includes commands for specifying integrity constraints.
 - **Authorization:** - The SQL DDL includes commands for specifying access rights to relations and views.

SQL Process



SQL Commands

◆ **DDL - Data Definition Language**

- **CREATE:** Creates a database, new table, a view of a table, or other object in the database.
- **ALTER:** Modifies an existing database object, such as a table.
- **DROP:** Deletes an entire table, a view of a table or other objects in the database.

◆ **DML - Data Manipulation Language**

- **SELECT:** Retrieves certain records from one or more tables.
- **INSERT:** Creates a record.
- **UPDATE:** Modifies records.
- **DELETE:** Deletes records.

SQL Commands

- ◆ **DCL - Data Control Language**
 - **GRANT:** Gives a privilege to user.
 - **REVOKE:** Takes back privileges granted from user.

Basic Data Types in SQL

- ◆ **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). Formatted numbers can be declared by using DECIMAL(i, j)—or DEC(i, j) or NUMERIC(i, j)—where i , the *precision*, is the total number of decimal digits and j , the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

Basic Data Types in SQL

- ◆ **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. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive* (a distinction is made between uppercase and lowercase). For fixed length strings, a shorter string is padded with blank characters to the right. For example, if the value ‘Smith’ is for an attribute of type CHAR(10), it is padded with five blank characters to become ‘Smith’ if needed. Padded blanks are generally ignored when strings are compared.

Basic Data Types in SQL

- ◆ **Character-string** data types
- ◆ Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

Basic Data Types in SQL

- ◆ **Bit-string** data types are either of fixed length n —BIT(n)—or varying length— BIT VARYING(n), where n is the maximum number of bits. The default for n , the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B‘10101’.5 Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.

Basic Data Types in SQL

- ◆ **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.
- ◆ 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.

Null Value

- ◆ Each type may include a special value called the **null value**. A **null value** indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered, as we shall see shortly.

Specifying Constraints

- ◆ Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
- ◆ Constraints can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.

SQL Constraints

- ◆ **Not Null:** It disallows NULL as a valid value. Ensures a column in a table can not be null.
(SQL allows the use of null values to indicate that values either unknown or does not exist.)
- ◆ **Default <Value> :** It specifies default value of an attribute. Without default clause, the default value is null for those attributes without having Not Null constraint.
- ◆ **Primary Key:** It specifies one or more attributes as a primary key and uniquely identifies each row/record in a database table.
- ◆ **Foreign Key:** It ensures referential integrity. A referential integrity constraint can be violated when tuples are inserted or deleted or when a primary or foreign key attribute value is modified. The default action for integrity violation is to reject the update operation that leads to violation. However one can specify a referential triggered action clause to any foreign key constraint. The option include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE.

SQL Constraints

- ◆ **Check:** It allows to specify a predicate that must be satisfied by any value assigned to an attribute.
E.g. Dnumber INT Not Null Check (Dnumber>0)
- ◆ **Index:** Used to create and retrieve data from the database very quickly.
- ◆ **Unique:** Ensures that all the values in a column are different. The **unique specification says that attributes Aj_1, Aj_2, \dots, Aj_m form a candidate key**; that is, no two tuples in the relation can be equal on all the listed attributes. However, candidate key attributes are permitted to be null unless they have explicitly been declared to be **not null**.

E.g. **unique (Aj_1, Aj_2, \dots, Aj_m)**

SQL Constraints

◆ Constraints for Referential Actions

- CASCADE: Delete or update the row from the parent table, and automatically delete or update the matching rows in the child table. Both ON DELETE CASCADE and ON UPDATE CASCADE are supported.
- SET NULL: Delete or update the row from the parent table, and set the foreign key column or columns in the child table to NULL.
- RESTRICT: Rejects the delete or update operation for the parent table.
- NO ACTION: Rejects the delete or update operation for the parent table if there is a related foreign key value in the referenced table. A keyword from standard SQL. In MySQL, equivalent to RESTRICT.
- SET DEFAULT: sets the referenced value to some default on the delete or update operation.

SQL Data Definition

- ◆ The set of relations in a database must be specified to the system by means of a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:
 - The schema for each relation.
 - The types of values associated with each attribute.
 - The integrity constraints.
 - The set of indices to be maintained for each relation.
 - The security and authorization information for each relation.
 - The physical storage structure of each relation on disk.

Basic SQL DDL Commands

- ◆ **For Creating Database Schema:**

- CREATE SCHEMA <database-name>AUTHORIZATION <user-identifier>;

- For Example:- Create Schema Employee Authorization Jagdish;

- ◆ **For Creating Tables:** Create table command is used to specify a new relation by giving its name and specifying its attributes , their data types and initial constraints, if any. The syntax is as follows:

- CREATE TABLE schema].table_name [{column descriptions}];
 - CREATE TABLE table_name [{column descriptions}];

- ◆ **For Creating Domains:** Create domain command is used to create a domain of particular data type. The syntax is;

- CREATE DOMAIN domain_name AS data type

Basic SQL DDL Commands

- ◆ **For Databases:**
- ◆ The database is created using create database statement.

For Example:- CREATE DATABASE COMPANY;

- ◆ **Using the Databases:**
- ◆ The database is used before creating or accessing tables within it.

For Example:- USE COMPANY;

- ◆ **For Domains:**
- ◆ The domain is created using create domain statement.

For Example:- CREATE DOMAIN SSN_TYPE AS CHAR(9);

- ◆ We can use SSN_TYPE in place of CHAR(9). (This feature may not be available in some implementations of SQL)

Basic SQL DDL Commands

- ◆ For Tables:
- ◆ The table created through create table statements are called base tables (base relations) means that the relation & its tuples are actually created & stored as a file by DBMS. The tables created by VIEW are virtual tables and may not correspond to any physical table

For Example:- **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));
```

Basic SQL DDL Commands: Referencing

For Example:- CREATE TABLE Employee

```
(    SSN      varchar(10) NOT NULL,
    Fname     varchar(20) NOT NULL,
    Lname     varchar(20) NOT NULL,
    Bdate     date,
    Address   varchar(30),
    Sex       char,
    Salary    decimal(10,2),
    SuperSSN  varchar(10),
    Dno       INT      NOT NULL,
PRIMARY KEY (SSN),
FOREIGN KEY (SuperSSN) REFERENCES Employee(SSN),
FOREIGN KEY (Dno) REFERENCES Department(Dnumber)
);
```

Basic SQL DDL Commands

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) ;

Basic SQL DDL Commands

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));

Basic SQL DDL Commands: Specifying Constraints

```
CREATE TABLE EMPLOYEE
(
    ...,
    Dno      INT      NOT NULL      DEFAULT 1,
    CONSTRAINT EMPPK
        PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                                ON DELETE SET NULL      ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
        FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                                ON DELETE SET DEFAULT  ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
(
    ...,
    Mgr_ssn CHAR(9)      NOT NULL      DEFAULT '888665555',
    ...,
    CONSTRAINT DEPTPK
        PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
        UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                                ON DELETE SET DEFAULT  ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
(
    ...,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                                ON DELETE CASCADE      ON UPDATE CASCADE);
```

Basic SQL DDL Commands: Specifying Constraints

- ◆ Using if not exists in create query:

```
CREATE TABLE IF NOT EXISTS newbook
( book_id varchar(15) NOT NULL UNIQUE,
book_name varchar(50) ,
isbn_no varchar(15) NOT NULL UNIQUE ,
cate_id varchar(8) ,
aut_id varchar(8) ,
pub_id varchar(8) ,
book_price decimal(8,2) ,
PRIMARY KEY (book_id) );
```

Basic SQL DDL Commands: Specifying Constraints

- ◆ Using if not exists in create query and check constraint:

```
CREATE TABLE IF NOT EXISTS parts (
    part_no VARCHAR(18) PRIMARY KEY,
    description VARCHAR(40),
    cost DECIMAL(10 , 2 ) NOT NULL CHECK(cost > 0),
    price DECIMAL (10,2) NOT NULL
);
```

- ◆ Using auto increment:

```
CREATE TABLE IF NOT EXISTS newauthor (
    id int NOT NULL AUTO_INCREMENT,
    aut_id varchar(8),
    aut_name varchar(50),
    country varchar(25),
    home_city varchar(25) NOT NULL,
    PRIMARY KEY (id));
```

Basic SQL DDL Commands: Specifying Constraints

- ◆ **Multiple Primary keys:**

```
CREATE TABLE IF NOT EXISTS newauthor(  
    aut_id varchar(8) NOT NULL ,  
    aut_name varchar(50) NOT NULL,  
    country varchar(25) NOT NULL,  
    home_city varchar(25) NOT NULL,  
    PRIMARY KEY (aut_id, home_city));
```

- ◆ **Using CHECK constraint using IN:**

```
CREATE TABLE IF NOT EXISTS newauthor(  
    aut_id varchar(8) NOT NULL ,  
    aut_name varchar(50) NOT NULL,  
    country varchar(25) NOT NULL CHECK (country IN ('USA','UK','Nepal')),  
    home_city varchar(25) NOT NULL,  
    PRIMARY KEY (aut_id,home_city));
```

Basic SQL DDL Commands: Specifying Constraints

- ◆ **Using CHECK constraint with LIKE:**

```
CREATE TABLE IF NOT EXISTS newbook
( book_id varchar(15) NOT NULL UNIQUE,
book_name varchar(50) ,
isbn_no varchar(15) NOT NULL UNIQUE ,
cate_id varchar(8) ,
aut_id varchar(8) ,
pub_id varchar(8) ,
dt_of_pub date CHECK (dt_of_pub LIKE '--/--/----'),
pub_lang varchar(15) ,
no_page decimal(5,0) CHECK(no_page>0) ,
book_price decimal(8,2) ,
PRIMARY KEY (book_id) );
```

Statements for Changing the schema

- ◆ **Drop Command:** Drop command can be used to drop the schema elements as tables, domains, or constraints. One can also drop a schema itself.
- ◆ **Schema/Database Deletion:**

DROP SCHEMA Schema_name [CASCADE / RESTRICT]

DROP DATABASE [IF EXISTS] { database_name }

This statement drops the schema. If cascade is used, then all tables, domains, and other elements are also deleted along with the schema. While if restrict option is used, then the schema is dropped only if it has no elements in it.

- E.g. DROP SCHEMA Company CASCADE;
- DROP DATABASE Company;
- DROP DATABASE IF EXISTS Company;

Statements for Changing the schema

- ◆ **Drop Command:** Drop command can be used to drop the schema elements as tables, domains, or constraints. One can also drop a schema itself.
- ◆ **Table Deletion:**

DROP TABLE Table_name [CASCADE / RESTRICT]

This statement drops the table. If restrict is used, then the table is dropped only if it is not referenced in any constraint. While if Cascade option is used, all the constraints & views that references the table are dropped automatically from the schema along with the table itself.

- E.g. DROP TABLE Employee CASCADE;

The Alter Command

- ◆ The alter command is used to change the definition of the base table and other schema elements. For base tables, the possible alter table actions include adding or dropping a column(attribute), changing a column definition, adding or dropping table constraints.
 - **To add an attribute:** *ALTER TABLE Table_name ADD [Column_name]*
E.g.: *ALTER TABLE Employee ADD Jobtype varchar(30);*
 - **To drop an attribute/column:** To drop a column, we can use CASCADE or RESTRICT option. With Cascade, all constraints, 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 reference the column. The Syntax is:
ALTER TABLE Table_name DROP [Column_name] [Cascade/Restrict]
E.g.: *ALTER TABLE Employee DROP Address CASCADE;*
 - **To set or drop default value of an attribute:**
E.g.: *ALTER TABLE Department ALTER MgrSSN DROP DEFAULT;*
ALTER TABLE Department ALTER MgrSSN SET DEFAULT “111”;

The Alter Command

- **To Add & Drop constraint:**

E.g.: CREATE TABLE Dependent

```
(  
    ESSN varchar(10),  
    Dependent_name varchar(30) CONSTRAINT Name_Not_Null NOTNULL,  
    Sex char,  
    Relationship varchar(10) CONSTRAINT Rel_None DEFAULT 'none',  
    CONSTRAINT DEPENDENT_PK PRIMARY KEY (ESSN, Dependent_name )  
);
```

Now to drop the constraint:

```
ALTER TABLE Dependent DROP CONSTRAINT Name_Not_Null ;
```

To add new constraint:

```
ALTER TABLE Dependent ADD CONSTRAINT DEP_FK FOREIGN KEY (ESSN)  
REFERENCES Employee(SSN);
```

The Alter Command

- To Add & Drop index:

```
ALTER TABLE table_name ADD INDEX index_name (column_name);
```

```
ALTER TABLE table_name DROP INDEX index_name;
```

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;

DROP DATABASE COMPANY;

- ◆ 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 Jag drops the schema itself.

The Drop Command

- ◆ If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, we can perform;

DROP TABLE DEPENDENT CASCADE;

- ◆ If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced in any constraints (for example, by foreign key definitions in another relation)* or views or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.
- ◆ Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition*.
- ◆ If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command.

The Truncate Command

- ◆ The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.
- ◆ E.g.: TRUNCATE TABLE Employee;
- ◆ You need not have to specify a WHERE clause in a TRUNCATE TABLE statement.
- ◆ TRUNCATE TABLE cannot be used when a foreign key references the table to be truncated, since TRUNCATE statements do not fire triggers. This could result in inconsistent data because ON DELETE/UPDATE triggers would not fire.

Basic Retrieval Queries in SQL

DML Statements in SQL

- ◆ **Select Statement:**
- ◆ SQL has one basic statement for retrieving information from a database; the SELECT statement.
- ◆ The basic form of Select statement is formed three clauses SELECT, FROM, and WHERE and has following structure:

SELECT <Attribute List>

FROM <Table List>

WHERE <Condition>

- <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.

DML Statements in SQL

- ◆ **Select Statement:**
- ◆ The SELECT clause corresponds to project operation of the relational algebra. It is used to list attributes desired in the result of query.
- ◆ The FROM clause corresponds to Cartesian-product of the relational algebra. It lists the relations to be scanned in the evaluation of expression.
- ◆ The WHERE clause corresponds to the selection predicate of the relation algebra. It consist of a predicate involving attributes of the relations that appear from clause.
- ◆ Example:

```
SELECT SSN, Fname, Lname  
FROM Employee  
WHERE Address= 'Kathmandu';
```

DML Statements in SQL

- ◆ **Select Statement:**
- ◆ A typical select query has the form

Select A_1, \dots, A_n

From $r_1, r_2, r_3, \dots, r_m$

Where P

This query is equivalent to following relational algebra query;

$$\pi_{A_1, \dots, A_n}(\sigma_P(r_1 \times r_2 \times r_3 \times \dots \times r_m))$$

- ◆ The result of the select query *may contain* duplicate tuples. To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used.

```
SELECT DISTINCT Fname  
FROM Employee  
WHERE Address= 'Kathmandu';
```

Ambiguous Attribute Name

- ◆ Some multi table queries may have same attribute name, so accessing them can be **ambiguous**. Thus to prevent this **ambiguity**, *prefixing* the relation name to the attribute name and separating the two by a period is done.
- ◆ Example: For each employee, retrieve the employee's name, and the name of his or her department.

Consider the schemas are

EMPLOYEE(Name, SSN, Address, Dob, Dno)

DEPARTMENT(Name, Dnumber)

SELECT Employee.Name, Department.Name

FROM Employee, Department

WHERE Employee.Dno=Department.Dnumber

- ◆ Here both employee and department have same attribute name “Name” so this forms ambiguous attribute, which can be accessed as above.

Aliasing in SQL

- ◆ Some queries need to refer to the same relation twice. In this case, *aliases* are given to the relation name.
- ◆ Example: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname  
FROM Employee AS E, Employee AS S  
WHERE E.SUPERSSN = S.SSN
```

- In above query, the alternate relation names E and S are called *aliases* or *tuple variables* for the EMPLOYEE relation.
- We can think of E and S as two *different copies* of EMPLOYEE; E represents employees in role of *supervisees* and S represents employees in role of *supervisors*.

Aliasing in SQL

- ◆ We can also rename the table names to shorter names by creating an *alias* for each table name to avoid repeated typing of long table names.
- ◆ Example: For each employee, retrieve the employee's names.

```
SELECT E.Fname, E.Lname  
FROM Employee AS E
```

- ◆ In above query, rather than typing Employee again and again we can use the alias E, which is shorter.

Aliasing and Renaming of Attribute in SQL

- ◆ It is also possible to **alias attributes** in the result of a query.

```
SELECT SSN as Employee_SSN  
FROM Employee  
WHERE Employee.Fname="John"
```

- ◆ It is also possible to **rename the relation and attributes** within the query in SQL by giving them aliases.

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

- ◆ Here, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on. The above query results all of records from Employee where the resulted records will have attribute names Fn, Mi, Ln, and

Unspecified Where clause & Use of *

- ◆ A *missing WHERE-clause* indicates no condition; hence, *all tuples* of the relations in the FROM-clause are selected.
- ◆ Example: Retrieve the SSN values for all employees.

```
SELECT SSN  
      FROM Employee
```

- ◆ If more than one relation is specified in the FROM-clause *and* there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected.
- ◆ Example: Select all combinations of Employee SSN and Department Name.

```
SELECT SSN, Dname  
      FROM Employee, Department
```

Unspecified Where clause & Use of *

- ◆ To retrieve all the attribute values of the selected tuples, a * is used, which stands for *all the attributes*.
- ◆ Example: Retrieve all the attribute values of any Employee who work in department number 5.

```
SELECT *
FROM Employee
WHERE DNO=5
```

- ◆ Example: Retrieve all the attributes of Employee & Department in which every employee works for “Research” department.

```
SELECT *
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME='Research' AND DNO=DNUMBER
```

Tables as Set 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. SQL does not automatically eliminate duplicate tuples in the results of queries.
- ◆ If we *do 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.
- ◆ **For Example:** Retrieve the salary of every employee.

SELECT ALL Salary FROM EMPLOYEE;

- ◆ **For Example:** Retrieve the distinct salary values of employee.

SELECT DISTINCT Salary FROM EMPLOYEE;

Set Operations

- ◆ SQL has directly incorporated some set operations. They are union operation (**UNION**), set difference (**MINUS/EXCEPT**) and intersection (**INTERSECT**). The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result.*
- ◆ The set operations apply only to *union compatible relations* ; the two relations must have the same attributes and the attributes must appear in the same order.
- ◆ Example:
 - `SELECT DISTINCT Fname
 FROM Employee
 WHERE Salary > 300000.00
UNION
 SELECT DISTINCT Fname
 FROM Employee
 WHERE Salary < 24000.0`

Substring Comparision

- ◆ The most commonly used operation on strings is pattern matching using the operator LIKE. We describe patterns by using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- ◆ Examples:
 - ‘Perry%’ matches any string beginning with “Perry”.
 - ‘%Perry’ matches any string ending with “Perry”.
 - ‘%Perry%’ matches any string containing “Perry” as a substring.
 - ‘---’ matches any string of exactly three characters.
 - ‘---%’ matches any string of at least three characters.
- ◆ Example: Retrieve all employee whose first name consist “Arun”.

```
SELECT *
  FROM Employee
 WHERE Fname LIKE '%Arun%';
```

Arithmetic Operators

- ◆ Standard numeric operators like addition (+), subtraction (-), multiplication (*) and division (/) can be applied to attributes having numeric domains.
- ◆ Example: Retrieve the resulting salary if every employee having salary greater than 30000 is given a 10 percent raise in the salary.

```
SELECT 1.1 * SALARY  
FROM   EMPLOYEE  
WHERE  SALARY > 30000;
```

Between Comparison Operators

- ◆ Between clause can be used as a comparison operator to check the value in between some values.
- ◆ Retrieve all employees in department 5 whose salary is between \$30,000 and \$40,000.

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

- ◆ The condition (Salary **BETWEEN** 30000 **AND** 40000) is equivalent to the condition ((Salary \geq 30000) **AND** (Salary \leq 40000)).

Ordering of Query Results

- ◆ The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s).
- ◆ The default order is in ascending order of values
- ◆ We can specify the keyword **DESC** if we want a descending order; the keyword **ASC** can be used to explicitly specify ascending order, even though it is the default
- ◆ Example: Retrieve names of all the employees in ascending order of their first name.

```
SELECT Fname, Minit, Lname  
FROM Employee  
ORDER BY Fname;
```

- ◆ Example: Retrieve names of all the employees in descending order of their first name.

```
SELECT Fname, Minit, Lname  
FROM Employee  
ORDER BY Fname DESC;
```

Ordering of Query Results

- ◆ Example: Retrieve names of all the employees, in alphabetical order, working for “Research” department.

```
SELECT Fname, Lname  
FROM Employee, Department  
WHERE Dname='Research' AND Dno=Dnumber  
ORDER BY Fname;
```

NULL Values

- ◆ It is possible for tuples to have a null value, denoted by *null*, for some of their attributes.
- ◆ *Null* signifies an unknown value or that a value does not exist.
- ◆ The predicate **IS NULL** or **IS NOT NULL** can be used to check for null values. *So equality comparison is not appropriate .*
- ◆ The result of any arithmetic expression involving *null* is *null*.
- ◆ All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.
- ◆ Example: Retrieve the names of all employees who do not have supervisors.

```
SELECT Fname, Lname  
FROM Employee  
WHERE SuperSSN IS NULL
```

Nested Queries

- ♦ Nested queries are those in which with in the WHERE clause, there is a complete SELECT-FROM-WHERE statement.
- ♦ Example: Retrieve the name and address of all employees who work for the 'Research' department.

```
SELECT      Fname, Lname, Address
FROM        Empyoe
WHERE       DNO IN (SELECT  Dnumber
                     FROM    Department
                     WHERE   Dname='Research' )
```

- The outer query select an EMPLOYEE tuple if its DNO value is in the result of either nested query.
- ♦ The comparison operator **IN** compares a value v with a set (or multi-set) of values V, and evaluates to **TRUE** if v is one of the elements in V.
- ♦ In general, we can have several levels of nested queries.

Correlated Nested Queries

- ◆ A subquery that uses a correlation name from an outer query is called a **correlated subquery**.
- ◆ If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query*, the two queries are said to be *correlated*.
- ◆ The result of a correlated nested query is *different for each tuple (or combination of tuples) of the relation(s) the outer query*
- ◆ Example: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
SELECT E.Fname, E.Lname
FROM Employee AS E
WHERE E.SSN IN (SELECT
                  ESSN
                  FROM
                  WHERE
                        Dependent
                        ESSN=E.SSN AND
                        E.Fname=Dependent_name)
```

The EXISTS Function

- ◆ EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not.
- ◆ EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query.
- ◆ Example: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
SELECT Fname, Lname
FROM Employee AS E
WHERE EXISTS (SELECT *
               FROM Dependent
               WHERE E.SSN=ESSN AND
                     E.Fname=Dependent_name)
```

- Here for each *Employee tuple*, evaluate the nested query, which retrieves all *Dependent tuples* with the same employee number & name as the *Employee tuple*; if at least one tuple Exists in the result of the nested query, then select that *Employee tuple*.

The EXISTS Function

- ◆ In general, EXISTS(Q) returns TRUE if there is at least one tuple exists in result of the nested query Q, & it returns FALSE otherwise. On the other hand, NOT EXISTS returns TRUE if there are no tuples in the result of nested query Q, and it returns FALSE otherwise.
- ◆ Example: Retrieve the names of employees who have no dependents.

```
SELECT      Fname, Lname
FROM        Employee
WHERE       NOT EXISTS (SELECT *
                        FROM Dependent
                        WHERE SSN=ESSN)
```

- In above query, the correlated nested query retrieves all *Dependent tuples* related to an *Employee tuple*. If *none exist*, the EMPLOYEE tuple is selected.

Explicit Sets

- ◆ It is also possible to use an **explicit (enumerated) set of values** in the WHERE-clause rather than a nested query
- ◆ Example: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

```
SELECT DISTINCT ESSN  
FROM WORKS_ON  
WHERE PNO IN (1, 2, 3);
```

- ◆ Example: Retrieve the social security numbers of all employees who DOES NOT work on project number 1, 2, or 3.

```
SELECT DISTINCT ESSN  
FROM WORKS_ON  
WHERE PNO NOT IN (1, 2, 3);
```

Set Comparision

- ♦ SQL also allows $< some, \leq some, \geq some, = some, and <> some$ comparisons.
- ♦ **SQL also allows $< all, \leq all, \geq all, = all, and <> all$ comparisons.**

Set Comparison

- ◆ As an example of the ability of a nested subquery to compare sets, consider the query **“Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.”**

```
SELECT distinct T.name
  FROM instructor as T, instructor as S
 WHERE T.salary > S.salary and S.dept_name = 'Biology';
```

- ◆ SQL does, however, offer an alternative style for writing the preceding query. The phrase **“greater than at least one”** is represented in SQL by $> \text{some}$.

```
SELECT name
  FROM instructor
 WHERE salary > some (SELECT salary
  FROM instructor
 WHERE dept_name = 'Biology');
```

Set Comparison

- ◆ The sub query `SELECT salary FROM instructor WHERE dept_name = 'Biology')`; generates the set of all salary values of all instructors in the Biology department.
- ◆ The *> some comparison in the where clause of the outer select is true if the salary value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.*

Set Comparison

- Let us find the names of all instructors that have a salary value greater than that of each instructor in the Biology department. The construct $> \text{all}$ corresponds to the phrase “greater than all.” Using this construct, we write the query as follows:

```
select name
      from instructor
     where salary > all (select salary
      from instructor
     where dept_name = 'Biology');
```

Set Comparison

- ◆ As another example of set comparisons, consider the query “**Find the departments that have the highest average salary.**”
- ◆ We begin by writing a query to find all average salaries, and then nest it as a subquery of a larger query that finds those departments for which the average salary is greater than or equal to all average salaries:

```
select dept_name
      from instructor
      group by dept_name
      having avg (salary) >= all (select avg (salary)
      from instructor
      group by dept_name);
```

Sub Queries in From Clause

- ♦ SQL allows a subquery expression to be used in the **from clause**. The key **concept** applied here is that any **select-from-where expression returns a relation as a result** and, therefore, can be inserted into another **select-from-where anywhere that a relation can appear**.

Sub Queries in From Clause

- ◆ Consider the query “Find the average instructors’ salaries of those departments where the average salary is greater than \$42,000.”
- ◆ The traditional query is ;

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

- ◆ The sub query in from clause is;

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
     where avg_salary > 42000;
```

INNER JOIN

- ◆ Equijoins may be specified by equating attribute names from two or more relations:

Example:

```
SELECT      FNAME, ADDRESS  
FROM        EMPLOYEE JOIN DEPARTMENT  
            ON DNO = DNUMBER  
WHERE       DEPARTMENT.DNAME='Research';
```

Or Equivalently,

```
SELECT      FNAME, ADDRESS  
FROM        EMPLOYEE INNER JOIN DEPARTMENT  
            ON DNO = DNUMBER  
WHERE       DEPARTMENT.DNAME='Research';
```

- ◆ The DNO attribute from EMPLOYEE is equated to DNUMBER attribute of DEPARTMENT for performing the join. The keyword INNER is optional

INNER JOIN

- ◆ Natural joins may be specified using the NATURAL JOIN construct. It automatically finds attributes having the same names for performing the join.
- ◆ Relations may be renamed to accommodate natural join using the AS construct.
- ◆ Example:

```
SELECT FNAME, LNAME, ADDRESS  
FROM (EMPLOYEE NATURAL JOIN DEPARTMENT  
      AS DEPT(DNAME, DNO, MSSN, MSDATE)  
WHERE DNAME='Research';
```

INNER JOIN

```
SELECT name, course id
FROM instructor, teaches
WHERE instructor.ID= teaches.ID;
```

- ◆ This query can be written more concisely using the natural-join operation in SQL as:

```
SELECT name, course id
FROM instructor natural join teaches;
```

OUTER JOIN

- ◆ This query can be written more concisely using the natural-join operation in SQL as:

```
SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name  
FROM (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S  
ON E.Super_ssn = S.Ssn);
```

```
SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name  
FROM (EMPLOYEE AS E RIGHT OUTER JOIN EMPLOYEE AS S  
ON E.Super_ssn = S.Ssn);
```

```
SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name  
FROM (EMPLOYEE AS E FULL OUTER JOIN EMPLOYEE AS S  
ON E.Super_ssn = S.Ssn);
```

OUTER JOIN

- ◆ If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN).
- ◆ Consider;

Student(sid, fname, lname, cid)

Course(cid, cname, credits)

```
SELECT *
FROM Student NATURAL LEFT OUTER JOIN Course;
```

```
SELECT *
FROM Student NATURAL RIGHT OUTER JOIN Course;
```

```
SELECT *
FROM Student NATURAL FULL OUTER JOIN Course;
```

MULTIWAY JOIN

- ◆ It is also possible to *nest join specifications*; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**.

MULTIWAY JOIN

- ◆ Consider, 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.
- ◆ The simple query is;

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

- ◆ The multiway join is;
- ```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
JOIN EMPLOYEE ON Mgr_ssn = Ssn)
WHERE Plocation = ‘Stafford’;
```

# Various OUTER JOIN Notations

- ◆ Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators  $+ =$ ,  $= +$ , and  $+ = +$  for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in Oracle.

# CROSS JOIN

- ◆ Cross Join is Cartesian Product of two relations. A cross join or cartesian product is formed when every row from one table is joined to all rows in another.

```
SELECT FNAME, ADDRESS
FROM EMPLOYEE CROSS JOIN DEPARTMENT
```

- ◆ When CROSS JOIN is executed with WHERE clause then it is similar to the INNER JOIN with ON clause.

# Aggregate Functions

- ◆ These functions operate on a collection (a set or multiset) of values of a column of a relation as input and return a single value. They include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.
- ◆ Example: Find the maximum salary and the average salary among all employees.

```
SELECT MAX(Salary), AVG(Salary)
FROM Employee
```

```
SELECT SUM (Salary) AS Total_Sal, MAX (Salary) AS Highest_Sal,
MIN (Salary) AS Lowest_Sal, AVG (Salary) AS Average_Sal
FROM EMPLOYEE;
```

- ◆ Retrieve the total number of employees in the 'Research' department.

```
SELECT COUNT (*)
FROM Employee, Department
WHERE Dno = Dnumber AND Dname='Research'
```

# Aggregate Functions

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

- ◆ However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(\*) because tuples instead of values are counted.
- ◆ Count the number of distinct salary values in the database.

```
SELECT COUNT (DISTINCT Salary)
FROM EMPLOYEE
```

# Group By Clause

- ◆ In many cases, we want to apply the aggregate functions *to* subgroups of tuples in a relation rather than all tuples. Each subgroup of tuples consists of the set of tuples that have *the same value* for the *grouping attribute(s)*. The function is applied to each subgroup independently.
- ◆ SQL has a **GROUP BY**-clause for specifying the grouping attributes, which must also appear in the *SELECT-clause*.
- ◆ Example: For each department, retrieve the department number and the number of employees in the department.

```
SELECT Dno, COUNT (*)
FROM Employee
GROUP BY Dno
```

- ◆ *In above query, the Employee tuples are divided into groups-each group having the same value for the grouping attribute Dno. The COUNT function is applied to each such group of tuples separately.*

# Group By Clause

- ◆ If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute*. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result

# Having Clause

- ◆ Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*.
- ◆ The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples) .
- ◆ **Example:** **Return all departments having more than twenty employees, and show the number of employees and their average salary.**

```
SELECT Dno, COUNT(*), AVG(Salary)
FROM Employee
GROUP BY Dno
HAVING COUNT(*) > 20;
```

- ◆ If a **where clause** and **having clause** appear in the same query, SQL applies the predicate in the **where clause** first. Tuples satisfying the **where predicate** are then placed into groups by the **group by clause**. SQL then applies **having clause**, if it is present, to each group; it removes the groups that do not satisfy the **having clause predicate**. The **select clause** uses the remaining groups to generate tuples of the result of the query.

# Summary of Select SQL Queries

- ◆ A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

```
SELECT <attribute list>
FROM <table list>
[WHERE <condition>]
[GROUP BY <grouping attribute(s)>]
[HAVING <group condition>]
[ORDER BY <attribute list>]
```

# Summary of Select SQL Queries

- ◆ The SELECT-clause lists the attributes or functions to be retrieved
- ◆ The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- ◆ The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- ◆ GROUP BY specifies grouping attributes
- ◆ HAVING specifies a condition for selection of groups
- ◆ ORDER BY specifies an order for displaying the result of a query
- ◆ A query is evaluated by first applying the FROM-clause followed by the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause. Conceptually, ORDER BY is applied at the end to sort the query results.
- ◆ The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to ~~all the~~ selected tuples in a query without a GROUP BY clause.

# Modification of database

## ◆ Insert:

- It is used to add one or more tuples to a relation
- Attribute values should be listed in the same order as the attributes were specified in the CREATE TABLE command
- **Eg:**

INSERT INTO EMPLOYEE

VALUES (102, “Peter”, “Crouch”, ‘9-5-1973’, “Rajajinagar Bangalore”, ’M’, 300000, 100, 5);

*It Inserts an entire EMPLOYEE record with corresponding values.*

- Specific attributes can be populated by explicitly specifying them by name:

INSERT INTO EMPLOYEE(Fname,Address,Dno)

VALUES (“Arun”, “Pokhara”, 5);

The other attributes will get either NULL values or their DEFAULT values

# Modification of database

## ◆ Delete:

- Removes tuples from a relation.
- Includes a WHERE-clause to select the tuples to be deleted.
- The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE-clause.
- Tuples are deleted from only one table at a time (unless CASCADE is specified on a referential integrity constraint).
- A missing WHERE-clause specifies that all tuples in the relation are to be deleted; the table then becomes an empty table.

# Modification of database

- ◆ **Delete:**

- ◆ **Examples:**

- DELETE FROM EMPLOYEE  
WHERE LNAME='Brown'
- DELETE FROM EMPLOYEE  
WHERE SSN='123456789'
- DELETE FROM EMPLOYEE  
WHERE DNO IN (SELECT DNUMBER  
FROM DEPARTMENT  
WHERE DNAME='Research')
- DELETE FROM EMPLOYEE

# Modification of database

## ◆ Update:

- Used to modify attribute values of one or more selected tuples.
- A WHERE-clause selects the tuples to be modified.
- An additional SET-clause specifies the attributes to be modified and their new values.
- Each command modifies tuples *in the same relation*.
- Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

|        |                                  |
|--------|----------------------------------|
| UPDATE | Project                          |
| SET    | Plocation = 'Bellaire', Dnum = 5 |
| WHERE  | Pnumber=10                       |

# Modification of database

- ◆ Update:
- ◆ Example: Give all employees in the 'Research' department a 10% raise in salary.

```
UPDATE
SET
WHERE
```

```
EMPLOYEE
SALARY = SALARY *1.1
DNO IN (SELECT DNUMBER
FROM DEPARTMENT
WHERE DNAME='Research')
```

- ◆ In this request, the modified SALARY value depends on the original SALARY value in each tuple.
- ◆ The reference to the SALARY attribute on the right of = refers to the old SALARY value before modification.
- ◆ The reference to the SALARY attribute on the left of = refers to the new SALARY value after modification.

# Views

- ◆ 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. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.
- ◆ We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- ◆ To create a view we use the command:

**create view  $v$  as <query expression>;**

<query expression> is any legal expression

$v$  is the view name

# Views

- ◆ For Example:

Create View WORKS\_ON1 as

(Select Fname, Lname, Pname, Hours

From EMPLOYEE, PROJECT, WORKS\_ON

Where SSN=ESSN AND Pnumber=Pno

Group by Pname);

| WORKS_ON1 |       |       |       |
|-----------|-------|-------|-------|
| Fname     | Lname | Pname | Hours |

- ◆ We can specify SQL queries on a newly create view:  
SELECT Fname, lname from WORKS\_ON1  
WHERE Pname='ProductX';
- ◆ When no longer needed, a view can be dropped:  
DROP WORKS\_ON1;

# Views

- ◆ For Example:

```
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;
```

| DEPT_INFO |            |           |
|-----------|------------|-----------|
| Dept_name | No_of_emps | Total_sal |
|           |            |           |

- ◆ The above VIEW explicitly specifies new attribute names for the view DEPT\_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

# View Implementation

- ◆ The problem of how a 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 (submitted by the user) into a query on the underlying base tables.
- ◆ For the view;

**SELECT** Fname, Lname

**FROM** WORKS\_ON1

**WHERE** Pname = ‘ProductX’;

- ◆ The above query 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’;

# View Implementation

- ◆ The disadvantage of the **query modification** 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 this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied *to one of the defining base tables*. The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

# View Implementation

- ◆ Different strategies as to when a materialized view is updated are possible. The **immediate update** strategy updates a view as soon as the base tables are changed; the **lazy update** strategy updates the view when needed by a view query; and the **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

# Updating Views

- ◆ 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. Hence, the view does not have to be realized or materialized at the time of *view definition* but rather at the time when we *specify a query* on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept upto-date.

# Updating Views

- ◆ A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have default values specified.*
- ◆ Views defined on multiple tables using joins are generally not updatable.
- ◆ Views defined using grouping and aggregate functions are not updatable.

# Updating Views

- ◆ For Example:

Create View Emp\_name as  
(Select Fname, Lname  
From EMPLOYEE);

```
UPDATE Emp_name
SET Fname= 'Hari'
WHERE Lname = 'Shrestha';
```

# Dropping Views

- ♦ 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 as;

**DROP VIEW WORKS\_ON1;**

# **Database Management System (MDS 505)**

## **Jagdish Bhatta**

## Unit -3

# Database Constraints and Normalization

## Database Constraints

- ◆ Already covered in unit 2.
- ◆ Refer 2.1 Relational Model.

# Assertions

- ◆ ASSERTION, which can be used to specify additional types of constraints that are outside the scope of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity).
- ◆ Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- ◆ An **assertion** is a predicate expressing a condition we wish the database to always satisfy.
- ◆ Domain constraints, functional dependency and referential integrity are special forms of assertion.
- ◆ **Syntax:**

create assertion assertion-name check predicate

## Assertions

- ◆ For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK
(
 NOT EXISTS (SELECT *
 FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
 WHERE E.Salary>M.Salary AND E.Dno = D.Dnumber AND D.Mgr_ssn =
 M.Ssn)
);
```

## Assertions

- ◆ The constraint name `SALARY_CONSTRAINT` is followed by the keyword `CHECK`, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied.
- ◆ The constraint name can be used later to disable the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any `WHERE` clause condition can be used, but many constraints can be specified using the `EXISTS` and `NOT EXISTS` style of SQL conditions.
- ◆ Whenever some tuples in the database cause the condition of an `ASSERTION` statement to evaluate to `FALSE`, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.
- ◆ By including this query inside a `NOT EXISTS` clause, the assertion will specify that the result of this query must be empty so that the condition will always be `TRUE`. Thus, the assertion is violated if the result of the query is not empty. In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

## Assertions

- ◆ A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated* in a specific table. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases. The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*.
- ◆ On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.
- ◆ When an assertion is created, the system tests it for validity. If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated. This testing may result in significant overhead if the assertions are complex. Because of this, the **assert** should be used with great care. Some system developer omits support for general assertions or provides specialized form of assertions that are easier to test.

# Triggers

- ◆ In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL.

# Triggers

- ◆ Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.
- ◆ Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY\_VIOLATION,5 which will notify the supervisor.

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN (NEW.SALARY > (SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN))
INFORM_SUPERVISOR (NEW.Supervisor_ssN,
NEW.Ssn);
```

# Triggers

- ◆ The trigger is given the name **SALARY\_VIOLATION**, which can be used to remove or deactivate the trigger later. A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:
  - The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
  - The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.
  - The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure **INFORM\_SUPERVISOR**.

# Triggers

- ◆ The following AFTER trigger simply prints ‘In the After Trigger’ when the trigger is executed:

USE Family;

```
CREATE TRIGGER TriggerOne ON Person
AFTER INSERT
```

AS

```
PRINT 'In the After Trigger';
```

- ◆ With the AFTER trigger enforced, the following code inserts a sample row:

```
INSERT Person(PersonID, LastName, FirstName, Gender)
VALUES (50, 'Ebob', 'Bill', 'M');
```

- ◆ Result:

**In the After Trigger**

# Stored Procedures

- ◆ A **stored procedure** (also termed **proc**, **storp**, **sproc**, **StoPro**, **StoredProc**, **StoreProc**, **sp**, or **SP**) is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary.
- ◆ Uses for stored procedures include data-validation (integrated into the database) or access-control mechanisms.
- ◆ Stored routines can be particularly useful in certain situations when multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.

# Stored Procedures

- ◆ **Create, alter and drop:**
- ◆ CREATE must be the first command in a batch; the termination of the batch ends the creation of the stored procedure. The following example creates a very simple stored procedure that retrieves data from the ProductCategory table :

```
CREATE PROCEDURE CategoryList
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM ProductCategory;
```

# Executing a Stored Procedure

- ◆ When calling a stored procedure within a SQL batch, the EXECUTE or CALL command executes the stored procedure with a few special rules. EXECUTE is typically coded as EXEC.
- ◆ For Eg.: EXEC/CALL dbo.CategoryList;

# Passing Data to Stored Procedure

- ◆ A stored procedure is more useful when it can be manipulated by parameters. The CategoryList stored procedure created previously returns all the product categories, but a procedure that performs a task on an individual row requires a method for passing the row ID to the procedure.

# Passing Data to Stored Procedure

- ◆ **Input parameters**
- ◆ You can add input parameters that pass data to the stored procedure by listing the parameters after the procedure name in the CREATE PROCEDURE command
- ◆ Each parameter must begin with an @ sign, and becomes a local variable within the procedure. Like local variables, the parameters must be defined with valid data types. When the stored procedure is called, the parameter must be included (unless the parameter has a default value).
- ◆ The following code sample creates a stored procedure that returns a single product category. The value passed by means of the parameter is available within the stored procedure as the variable @CategoryName in the WHERE clause:

# Passing Data to Stored Procedure

## ◆ Input parameters

```
CREATE PROCEDURE CategoryGet (@CategoryName VARCHAR(35))
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM ProductCategory
WHERE ProductCategoryName = @CategoryName;
```

# Passing Data to Stored Procedure

- When the following code sample is executed, the Unicode string literal ‘Kite’ is passed to the stored procedure and substituted for the variable in the WHERE clause:

```
EXEC CategoryGet 'Kite';
```

- Result:

| ProductCategoryName | ProductCategoryDescription                                             |
|---------------------|------------------------------------------------------------------------|
| Kite                | a variety of kites, from simple to stunt, to Chinese, to novelty kites |

# Passing Data to Stored Procedure

- ◆ If multiple parameters are involved, the parameter name can be specified in any order, or the parameter values listed in order. If the two methods are mixed, then as soon as the parameter is provided by name, all the following parameters must be as well.
- ◆ The next four examples each demonstrate calling a stored procedure and passing the parameters by original position and by name:

EXEC Schema.StoredProcedure

*@Parameter1 = n,*

*@Parameter2 = 'n';*

EXEC Schema.StoredProcedure

*@Parameter2 = 'n',*

*@Parameter1 = n;*

EXEC Schema.StoredProcedure *n, 'n';*

EXEC Schema.StoredProcedure *n, @Parameter2 = 'n';*

# Passing Data to Stored Procedure

- ◆ **Parameter defaults**
- ◆ You must supply every parameter when calling a stored procedure, unless that parameter has been created with a default value. You establish the default by appending an equal sign and the default to the parameter, as follows;

```
CREATE PROCEDURE StoredProcedure
(
 @Variable DataType = Default Value
)
```

# Passing Data to Stored Procedure

- ◆ **Parameter defaults**
- ◆ The following code, demonstrates a stored procedure default. If a product category name is passed in this stored procedure, the stored procedure returns only the selected product category. However, if nothing is passed, the NULL default is used in the WHERE clause to return all the product categories:

```
CREATE PROCEDURE pProductCategory_Fetch2
(@Search VARCHAR(50) = NULL)
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM ProductCategory
WHERE ProductCategoryName = @Search OR @Search IS NULL;
```

# Passing Data to Stored Procedure

- ◆ Parameter defaults
- ◆ The first execution passes a product category:

```
EXEC dbo.pProductCategory_Fetch2 'OBX';
```

- ◆ Result:

Result:

| ProductCategoryName | ProductCategoryDescription |
|---------------------|----------------------------|
| OBX                 | OBX stuff                  |

- ◆ When pProductCategory\_Fetch executes without a parameter, the @Search parameter's default of NULL allows the WHERE clause to evaluate to true for every row, as follows:

```
EXEC dbo.pProductCategory_Fetch2;
```

| ProductCategoryName | ProductCategoryDescription                                             |
|---------------------|------------------------------------------------------------------------|
| Accessory           | kite flying accessories                                                |
| Book                | Outer Banks books                                                      |
| Clothing            | OBX t-shirts, hats, jackets                                            |
| Kite                | a variety of kites, from simple to stunt, to Chinese, to novelty kites |
| Material            | Kite construction material                                             |
| OBX                 | OBX stuff                                                              |
| Toy                 | Kids stuff                                                             |
| Video               | stunt kite contexts and lessons, and Outer Banks videos                |

# Returning Data from Stored Procedure

- ◆ **Output Parameter defaults**
- ◆ Output parameters enable a stored procedure to return data to the calling client procedure. The keyword OUTPUT is required both when the procedure is created and when it is called. Within the stored procedure, the output parameter appears as a local variable. In the calling procedure or batch, a variable must have been created to receive the output parameter. When the stored procedure concludes, its current value is passed to the calling procedure's local variable.
- ◆ **Eg:**

```
CREATE PROC GetProductName (
 @ProductCode CHAR(10),
 @ProductName VARCHAR(25) OUTPUT
)
AS
SELECT @ProductName = ProductName
FROM Product
WHERE Code = @ProductCode;
RETURN;
```

# Returning Data from Stored Procedure

- ◆ Output Parameter defaults

```
DECLARE @ProdName VARCHAR(25);
EXEC GetProductName '1001', @ProdName OUTPUT;
PRINT @ProdName;
```

- ◆ Result:  
Basic Box Kite 21 inch

## Informal Design Guidelines for Relational Databases

- ◆ The four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:
  - Making sure that the semantics of the attributes is clear in the schema
  - Reducing the redundant information in tuples
  - Reducing the NULL values in tuples
  - Disallowing the possibility of generating spurious tuples

# Informal Design Guidelines for Relational Databases

- ◆ **Imparting Clear Semantics to Attributes in Relations**
- ◆ 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.
- ◆ **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.

# Informal Design Guidelines for Relational Databases

- ◆ **Redundant Information in Tuples and Update Anomalies**
- ◆ One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space.
- ◆ Redundant information storage leads to extra space and chances of having anomalies.

# Informal Design Guidelines for Relational Databases

- ◆ **Redundant Information in Tuples and Update Anomalies**
- ◆ **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.

# Informal Design Guidelines for Relational Databases

## ◆ **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 and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical 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:

- The attribute *does not apply* to this tuple.
- The attribute value for this tuple is *unknown*.
- The value is *known but absent*; that is, it has not been recorded yet.

# Informal Design Guidelines for Relational Databases

- ◆ **NULL Values in Tuples**
- ◆ Having the same representation for all NULLs compromises the different meanings they may have.
- ◆ **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.

# Informal Design Guidelines for Relational Databases

- ◆ **Generation of Spurious Tuples**
- ◆ Generation of unnecessary tuples while using joins, which actually does not exist in base relations.
  
- ◆ **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.

## Informal Design Guidelines for Relational Databases

- ◆ Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- ◆ Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- ◆ Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

# Functional Dependency

- ◆ Functional dependency is a constraint between two sets of attributes from the database. The functional dependency between two sets of attributes  $X$  &  $Y$  in a relation  $R$  is a constraint on possible tuples that can form a relation instance  $r$  of  $R$ . This constraint is that for any two tuples  $t_1$  &  $t_2$  in  $r$  if  $t_1[X]=t_2[X]$  then  $t_1[Y]=t_2[Y]$ . This functional dependency is represented by  $X \rightarrow Y$ .
- ◆ This means that the values of the  $Y$  component of a tuple in  $r$  depend on, or are *determined by*, the values of the  $X$  component; alternatively, the values of the  $X$  component of a tuple uniquely (or **functionally**) *determine* the values of the  $Y$  component. We also say that there is a 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**.

# Functional Dependency

- ◆ This functional dependency generalizes the concept of key. If K is a key of R, then K functionally determines all attributes in R (since we never have two distinct tuples with  $t_1[K]=t_2[K]$ )
- ◆ A functional dependency is a property of the **semantics or meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of  $R$ —that is, how they relate to one another—to specify the functional dependencies that should hold on *all* relation states (extensions)  $r$  of  $R$ . Relation extensions  $r(R)$  that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of  $R$ . Hence, the main use of functional dependencies is to describe further a relation schema  $R$  by specifying constraints on its attributes that must hold *at all times*.

# Functional Dependency

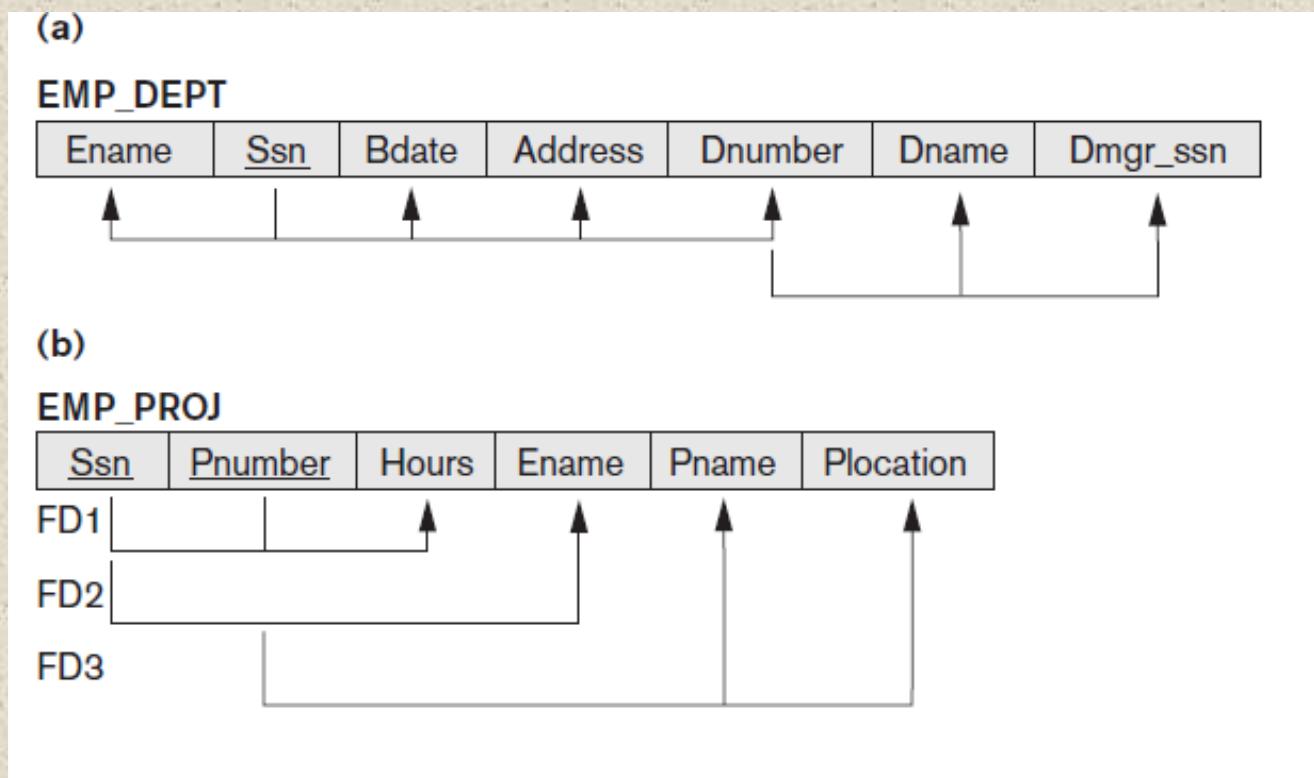
- ◆ **Trivial** – If a functional dependency (FD)  $X \rightarrow Y$  holds, where  $Y$  is a subset of  $X$ , then it is called a trivial FD. Trivial FDs always hold.
- ◆ **Non-trivial** – If an FD  $X \rightarrow Y$  holds, where  $Y$  is not a subset of  $X$ , then it is called a non-trivial FD.

# Examples of FD Constraints

- ◆ In the database schema COMPANY;
  - social security number determines employee name  
 $SSN \rightarrow ENAME$
  - project number determines project name and location  
 $PNUMBER \rightarrow \{PNAME, PLOCATION\}$
  - employee ssn and project number determines the hours per week that the employee works on the project  
 $\{SSN, PNUMBER\} \rightarrow HOURS$
- ◆ 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  $R$ .

# Examples of FD Constraints

- Following figure introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, whereas the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.



# Inference Rules for FDs

- Given a set of FDs  $F$ , we can *infer* additional FDs that hold whenever the FDs in  $F$  hold

## Armstrong's inference rules:

IR1. (**Reflexive**) If  $Y$  subset-of  $X$ , then  $X \rightarrow Y$

IR2. (**Augmentation**) If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$

(Notation:  $XZ$  stands for  $X \cup Z$ )

IR3. (**Transitive**) If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$

- IR1, IR2, IR3 form a *sound* and *complete* set of inference rules. *Sound* means that from a given set of FDs  $F$  specified on relation schema  $R$ , any dependency that we can infer from  $F$  holds in every instance  $r$  of  $R$  that satisfies in  $F$  & *complete* means we can infer all possible dependencies from given set of FDs  $F$ .

# Inference Rules for FDs

Some additional inference rules that are useful:

**(Decomposition)** If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$

**(Union)** If  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$

**(Psuedotransitivity)** If  $X \rightarrow Y$  and  $WY \rightarrow Z$ , then  $WX \rightarrow Z$

# Inference Rules for FDs

- ◆ **Closure of a set F of FDs** is the set  $F^+$  of all FDs that can be inferred from F. Using the inference rules  $F^+$  can be formed.
- ◆ **Closure of a set of attributes X with respect to F** is the set  $X^+$  of all attributes that are functionally determined by X.

$X^+$  can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F

**Algorithm:** Determining  $X^+$ , the closure of X under F

$X^+ = X;$

repeat

$\text{old}X^+ = X^+;$

    for each functional dependency  $Y \rightarrow Z$  in F do

        if  $Y$  subset-of  $X^+$  then  $X^+ = X^+ \cup Z;$

until ( $X^+ = \text{old } X^+$ );

# Example: Closure of set of FDs

- ◆  $R = (A, B, C, G, H, I)$   
 $F = \{ A \rightarrow B$   
     $A \rightarrow C$   
     $CG \rightarrow H$   
     $CG \rightarrow I$   
     $B \rightarrow H\}$
- ◆ Some members of  $F^+$ 
  - $A \rightarrow H$ 
    - by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$
  - $AG \rightarrow I$ 
    - by augmenting  $A \rightarrow C$  with G, to get  $AG \rightarrow CG$  and then transitivity with  $CG \rightarrow I$
  - $CG \rightarrow HI$ 
    - from  $CG \rightarrow H$  and  $CG \rightarrow I$ : “union rule” can be inferred from

# Example: Closure of set of Attributes

- ◆  $R = (A, B, C, G, H, I)$
- ◆  $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- ◆ Now for  $(AG)^+$ , using the algorithm we have
  1.  $result = \{A, G\}$
  2.  $result = \{A, B, C, G\}$       ( $A \rightarrow C$  and  $A \rightarrow B$ )
  3.  $result = \{A, B, C, G, H\}$     ( $CG \rightarrow H$  and  $CG \subseteq ABCG$ )
  4.  $result = ABCGHI$       ( $CG \rightarrow I$  and  $CG \subseteq ABCGH$ )

*And so on...*

# Equivalence of sets of FDs

- ◆ Two sets of FDs F and G are **equivalent** if:
  - every FD in F can be inferred from G, *and*
  - every FD in G can be inferred from F
- ◆ Hence, F and G are equivalent if  $F^+ = G^+$   
i.e. if both of the conditions F covers G and G covers F holds.
- ◆ A set of functional dependencies F is said to cover another set of functional dependencies G, F *covers* G, if every FD in G can be inferred from F (i.e., if  $G^+ \subseteq F^+$ ).

# Normalization of Relations

- ◆ **Normalization:** The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations.
- ◆ Normalization of data can be looked upon as 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.
- ◆ **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- ◆ We have 1NF, 2NF, 3NF, BCNF based on keys and FDs of a relation schema and 4NF based on keys, multi-valued dependencies : MVDs; 5NF based on keys, join dependencies : JDs.

# Importance of Normalization

- ◆ **Data consistency:** Data consistency means that the data is always real and it is not ambiguous.
- ◆ **Data becomes nonredundant:** Non-redundant means that only one copy of data is available for each user and for every time. There are no multiple copies of the same data for different persons. So when data is changed in one file and stay in one file. Then of course data is consistent and non-redundant. Here redundant is not the same as a backup of data, both are different things.

# Importance of Normalization

- ◆ **Reduce insertion, deletion and updating anomalies:**
  - **Insertion anomaly** is an anomaly that occurs when we want to insert data into the database but the data is not completely or correctly inserted in the target attributes. If completely inserted in the database then not correctly entered.
  - **Deletion anomaly** is an anomaly that occurs when we want to delete data in the database but the data is not completely or correctly deleted in the target attributes.
  - **Update anomaly** is an anomaly that occurs when we want to update data in the database but the data is not completely or correctly updated in the target attributes.

# Importance of Normalization

- ◆ **Database table compaction:** When we normalize the database, we convert the large table into a smaller table that leads to data and table compaction. Compaction means to have the least and required size.
- ◆ **Isolation of Data:** A good designed database states that the changes in one table or field do not affect other. This is achieved through Normalization.
- ◆ **Better performance**

## First Normal Form(1NF)

- ◆ It states that domain of an attribute must include only atomic values.
- ◆ Disallows composite attributes, multivalued attributes and **nested relations**; attributes whose values *for an individual tuple* are non-atomic.
- ◆ It is considered to be part of the definition of relation

## 1NF: Example

- Following example is not in normal form as attribute dlocation is multivalued.

(a)

| DEPARTMENT |                |         |            |
|------------|----------------|---------|------------|
| DNAME      | <u>DNUMBER</u> | DMGRSSN | DLOCATIONS |
|            |                |         |            |

(b)

| DNAME          | <u>DNUMBER</u> | DMGRSSN   | DLOCATIONS                     |
|----------------|----------------|-----------|--------------------------------|
| Research       | 5              | 333445555 | (Bellaire, Sugarland, Houston) |
| Administration | 4              | 987654321 | (Stafford)                     |
| Headquarters   | 1              | 888665555 | (Houston)                      |

## 1NF:Example

- ◆ To resolve it into 1NF, remove the attribute DLocation that violates the 1NF & place it in a separate relation, Dept\_locations along with the primary key Dnumber of Department. The primary key of this relation is the combination {Dnumber, Dlocation}. Hence the normalized relations will be as:

DEPARTMENT (Dnumber, Dname, MgrSSN)

Dept\_locations (Dnumber, Dlocation)

# 1NF:Example (Normalizing nested relations into 1NF )

(a)

**EMP\_PROJ**

| SSN | ENAME | PROJS   |       |
|-----|-------|---------|-------|
|     |       | PNUMBER | HOURS |

(b)

**EMP\_PROJ**

| SSN       | ENAME               | PNUMBER | HOURS |
|-----------|---------------------|---------|-------|
| 123456789 | Smith,John B.       | 1       | 32.5  |
|           |                     | 2       | 7.5   |
| 666884444 | Narayan,Ramesh K.   | 3       | 40.0  |
|           |                     | 1       | 20.0  |
| 453453453 | English,Joyce A.    | 2       | 20.0  |
|           |                     | 1       | 20.0  |
| 333445555 | Wong,Franklin T.    | 2       | 10.0  |
|           |                     | 3       | 10.0  |
|           |                     | 10      | 10.0  |
|           |                     | 20      | 10.0  |
| 999887777 | Zelaya,Alicia J.    | 30      | 30.0  |
|           |                     | 10      | 10.0  |
| 987987987 | Jabbar,Ahmad V.     | 10      | 35.0  |
|           |                     | 30      | 5.0   |
| 987654321 | Wallace,Jennifer S. | 30      | 20.0  |
|           |                     | 20      | 15.0  |
| 888665555 | Borg,James E.       | 20      | null  |

(c)

**EMP\_PROJ1**

| SSN | ENAME |
|-----|-------|
|-----|-------|

**EMP\_PROJ2**

| SSN | PNUMBER | HOURS |
|-----|---------|-------|
|-----|---------|-------|

Decomposing **EMP\_PROJ** into 1NF relations **EMP\_PROJ1** and **EMP\_PROJ2** by propagating the primary key.

## Second Normal Form(2NF)

- ◆ A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on the primary key.
- ◆ R can be decomposed into 2NF relations via the process of 2NF normalization.
  
- ◆ **Prime attribute** - attribute that is member of the primary key K
- ◆ **Full functional dependency** - a FD  $Y \rightarrow Z$  where removal of any attribute from Y means the FD does not hold any more

Examples:- {SSN, PNUMBER}  $\rightarrow$  HOURS is a full FD since neither SSN  $\rightarrow$  HOURS nor PNUMBER  $\rightarrow$  HOURS hold

- {SSN, PNUMBER}  $\rightarrow$  ENAME is *not* a full FD (it is called a *partial dependency*) since SSN  $\rightarrow$  ENAME also holds

## 2NF: Example

(a)

EMP\_PROJ

| SSN | PNUMBER | HOURS | ENAME | PNAME | PLOCATION |
|-----|---------|-------|-------|-------|-----------|
| FD1 |         |       |       |       |           |
| FD2 |         |       |       |       |           |
| FD3 |         |       |       |       |           |

2NF NORMALIZATION



EP1

| SSN | PNUMBER | HOURS |
|-----|---------|-------|
| FD1 |         |       |

EP2

| SSN | ENAME |
|-----|-------|
| FD2 |       |

EP3

| PNUMBER | PNAME | PLOCATION |
|---------|-------|-----------|
| FD3     |       |           |

## Third Normal Form(3NF)

- ◆ **Transitive functional dependency** - a FD  $X \rightarrow Z$  that can be derived from two FDs  $X \rightarrow Y$  and  $Y \rightarrow Z$

Examples:

$SSN \rightarrow DMGRSSN$  is a *transitive* FD since

$SSN \rightarrow DNUMBER$  and  $DNUMBER \rightarrow DMGRSSN$  hold

$SSN \rightarrow ENAME$  is *non-transitive* since there is no set of attributes  $X$  where  $SSN \rightarrow X$  and  $X \rightarrow ENAME$

## Third Normal Form(3NF)

- ◆ A relation schema R is in **third normal form (3NF)** if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key
- ◆ R can be decomposed into 3NF relations via the process of 3NF normalization

### **NOTE:**

In  $X \rightarrow Y$  and  $Y \rightarrow Z$ , with X as the primary key, we consider this a problem only if Y is not a candidate key. When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary ).

Here,  $SSN \rightarrow Emp\# \rightarrow Salary$  and Emp# is a candidate key.

## 3NF: Example

(b)

EMP\_DEPT

| ENAME | <u>SSN</u> | BDATE | ADDRESS | DNUMBER | DNAME | DMGRSSN |
|-------|------------|-------|---------|---------|-------|---------|
|       |            |       |         |         |       |         |



ED1

| ENAME | <u>SSN</u> | BDATE | ADDRESS | DNUMBER |
|-------|------------|-------|---------|---------|
|       |            |       |         |         |

ED2

| DNUMBER | DNAME | DMGRSSN |
|---------|-------|---------|
|         |       |         |

# Summary of 1NF, 2NF, 3NF

| Normal Forms | Test                                                                                                                                             | Remedy(Normalization)                                                                                                                                                                                              |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1NF          | Relation should have no nonatomic attributes or nested relations.                                                                                | Form new relations for each nonatomic attribute or nested relation.                                                                                                                                                |
| 2NF          | For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key. | Decompose & set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it. |

## Summary of 1NF, 2NF, 3NF

| Normal Forms | Test                                                                                                                                                                                                                                   | Remedy(Normalization)                                                                                                                    |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| 3NF          | <p>Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.</p> | <p>Decompose &amp; set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).</p> |

## Boyce-Codd Normal Form (BCNF)

- ◆ A relation schema  $R$  is in **Boyce-Codd Normal Form (BCNF)** if whenever any **non-trivial functional dependency**  $X \rightarrow A$  holds in  $R$ , then  $X$  is a superkey of  $R$ .
- ◆ BCNF was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. i.e every relation in BCNF is also in 3NF ;however, a relation in 3NF is not necessarily in BCNF. Thus a relation in BCNF should be in 3NF.
- ◆ In practice, most relation schemas that are in 3NF are also in BCNF. Only if there exists some functional dependency  $X \rightarrow A$  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.

# Boyce-Codd Normal Form (BCNF)

- ◆ Example: Relation TEACH with {student, course} as a candidate key and having following two dependencies;

**FD1:** {Student, Course} → Instructor    **FD2:** Instructor → Course

**Figure 14.13** A relation TEACH that is in 3NF but not in BCNF.

| TEACH   |                   |            |
|---------|-------------------|------------|
| STUDENT | COURSE            | INSTRUCTOR |
| Narayan | Database          | Mark       |
| Smith   | Database          | Navathe    |
| Smith   | Operating Systems | Ammar      |
| Smith   | Theory            | Schulman   |
| Wallace | Database          | Mark       |
| Wallace | Operating Systems | Ahamad     |
| Wong    | Database          | Omiecinski |
| Zelaya  | Database          | Navathe    |

## **Boyce-Codd Normal Form (BCNF)**

- ◆ The relation Teach is in 3NF but not in BCNF.
- ◆ A relation **NOT** in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations
- ◆ Three possible decompositions for relation TEACH
  1. {student, instructor} and {student, course}
  2. {course, instructor} and {course, student}
  3. {instructor, course} and {instructor, student}
- ◆ All three decompositions will lose FD1. We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- ◆ Out of the above three, only the **3<sup>rd</sup>** decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).

## **Boyce-Codd Normal Form (BCNF)**

- ◆ In general, a relation  $R$  not in BCNF can be decomposed so as to meet the nonadditive join property by the following procedure. It decomposes  $R$  successively into a set of relations that are in BCNF:
- ◆ Let  $R$  be the relation not in BCNF, let  $X \subseteq R$ , and let  $X \rightarrow A$  be the FD that causes a violation of BCNF.  $R$  may be decomposed into two relations:
  - $R - A$
  - $XA$
- ◆ If either  $R - A$  or  $XA$  is not in BCNF, repeat the process.
- ◆ Hence, in previous example in slide number 38, the proper decomposition of TEACH into BCNF relations is:

TEACH1 (Instructor, Course) and TEACH2 (Instructor, Student)

# Decomposition

- ◆ Consider a universal relation schema  $R = \{A_1, A_2, \dots, A_n\}$ .
- ◆ **Decomposition:** The process of decomposing the universal relation schema  $R$  into a set of relation schemas  $D = \{R_1, R_2, \dots, R_m\}$  that will become the relational database schema;  $D$  is called a *decomposition of  $R$* .
- ◆ Each attribute in  $R$  will appear in at least one relation schema  $R_i$  in the decomposition so that no attributes are “lost”.  
i.e  $R = R_1 \cup R_2 \cup R_3 \cup \dots \cup R_m$
- ◆ Another goal of decomposition is to have each individual relation  $R_i$  in the decomposition  $D$  be in BCNF or 3NF.
- ◆ Additional properties of decomposition are needed to prevent from generating spurious tuples

# Properties of Relational Decomposition

## ♦ Dependency Preservation Property of a Decomposition

### Definition:

- ♦ Given a set of dependencies  $F$  on  $R$ , the **projection** of  $F$  on  $R_i$ , denoted by  $\pi_{R_i}(F)$  where  $R_i$  is a subset of  $R$ , is the set of dependencies  $X \rightarrow Y$  in  $F^+$  such that the attributes in  $X \rightarrow Y$  are all contained in  $R_i$ . Hence, the projection of  $F$  on each relation schema  $R_i$  in the decomposition  $D$  is the set of functional dependencies in  $F^+$ , the closure of  $F$ , such that all their left- and right-hand-side attributes are in  $R_i$ .

# Properties of Relational Decomposition

- ◆ **Dependency Preservation Property of a Decomposition:**
  - a decomposition  $D = \{R_1, R_2, \dots, R_m\}$  of  $R$  is **dependency-preserving** with respect to  $F$  if the union of the projections of  $F$  on each  $R_i$  in  $D$  is equivalent to  $F$ ; that is,  $((\pi_{R1}(F)) \cup \dots \cup (\pi_{Rm}(F)))^+ = F^+$
  - If the decomposition is not dependency preserving some dependency is lost in the decomposition.

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

# Properties of Relational Decomposition

## ◆ Lossless (Nonadditive) Join Property:

- a decomposition  $D = \{R_1, R_2, \dots, R_m\}$  of  $R$  has the **lossless (nonadditive) join property** with respect to the set of dependencies  $F$  on  $R$  if, for *every* relation state  $r$  of  $R$  that satisfies  $F$ , the following holds,

$* (\pi_{R_1}(r), \dots, \pi_{R_m}(r)) = r$  ; where  $*$  is the natural join of all the relations in  $D$ :

- In other words, if we can recover original relation from decomposed relations then lossless property holds.

**Note:** The word **loss** in *lossless* refers to *loss of information*, not to loss of tuples. In fact, for “loss of information” a better term is “addition of spurious information”. So, if a decomposition does not have this property then we may get the spurious tuples after the project and natural join operations.

# Properties of Relational Decomposition

## ◆ Lossless (Nonadditive) Join Property:

- Thus a decomposition is lossless if the joining attribute is a superkey in at least one of the new relations obtained after decomposition.
- Formally,

A decomposition of  $R$  into  $R_1$  and  $R_2$  is lossless join if and only if at least one of the following dependencies is in  $F^+$ :

$$R_1 \cap R_2 \rightarrow R_1$$

$$R_1 \cap R_2 \rightarrow R_2$$

i.e.  $R_1 \cap R_2$  forms a superkey of either  $R_1$  or  $R_2$

# Multivalued Dependency

- Let  $R$  be a relation schema and let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The *multivalued dependency*

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

holds on  $R$  if in any legal relation  $r(R)$ , for all pairs for tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , there exist tuples  $t_3$  and  $t_4$  in  $r$  such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_3[Z] = t_2[Z]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_4[Z] = t_1[Z] ; \text{ where } Z = R - (\alpha \cup \beta)$$

A multivalued dependency can be trivial or non-trivial. It is trivial if  $\beta \subseteq \alpha$  or  $\alpha \cup \beta = R$ . Otherwise it is non-trivial.

# Multivalued Dependency

- ◆ A relation is said to have multi-valued dependency, if the following conditions are true;
  - For a dependency  $A \rightarrow B$ , if for a single value of A, multiple value of B exists, then the relation may have multi-valued dependency,  $A \twoheadrightarrow B$ .
  - Also, a relation should have at-least 3 columns for it to have a multi-valued dependency.
  - And, for a relation  $R(A,B,C)$ , if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.
- ◆ If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

# Multivalued Dependency

- Below we have a college enrolment table with columns SID, Course and Hobby.

| <u>SID</u> | <u>Course</u> | <u>Hobby</u> |
|------------|---------------|--------------|
| 1          | Science       | Cricket      |
| 1          | Maths         | Hockey       |
| 2          | DBMS          | Cricket      |
| 2          | PHP           | Hockey       |

- As you can see in the table above, student with SID **1** has opted for two courses, **Science** and **Maths**, and has two hobbies, **Cricket** and **Hockey**.
- Here it exists multivalued dependency  $\text{SID} \rightarrow\rightarrow \text{Course}$  &  $\text{SID} \rightarrow\rightarrow \text{Hobby}$

# Multivalued Dependency

- ◆ Example:

| Emp          |              |                       |
|--------------|--------------|-----------------------|
| <u>Ename</u> | <u>Pname</u> | <u>Dependent_name</u> |
| Ram          | X            | Rakesh                |
| Ram          | Y            | Rao                   |
| Ram          | X            | Rao                   |
| Ram          | Y            | Rakesh                |

- ◆ Here we have  $\text{Ename} \rightarrow\rightarrow \text{Pname}$  &  $\text{Ename} \rightarrow\rightarrow \text{Dependent\_name}$

## Fourth Normal Form(4NF)

- ◆ A relation schema  $R$  is in 4NF with respect to a set of dependencies  $F$  (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency  $X \rightarrow\!\!\rightarrow Y$  in  $F$ ,  $X$  is a superkey for  $R$ .
- ◆ Or, equivalently;
- ◆ A relation schema  $R$  is in 4NF with respect to a set  $D$  of functional and multivalued dependencies if for all multivalued dependencies in  $D^+$  of the form  $\alpha \rightarrow\!\!\rightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following hold:
  - $\alpha \rightarrow\!\!\rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$  or  $\alpha \cup \beta = R$ )
  - $\alpha$  is a superkey for schema  $R$
- ◆ If a relation is in 4NF it is in BCNF.

## Fourth Normal Form(4NF)

- ◆ 4NF is violated when a relation has undesirable multivalued dependencies and hence can be used to identify and decompose such relations.
- ◆ For a relation to be in 4NF, it should satisfy the following two conditions:
  - It should be in the **Boyce-Codd Normal Form**.
  - It should not have any **Multi-valued Dependency**.

# 4NF: Example

- ◆ Example:

| <u>SID</u> | <u>Course</u> | <u>Hobby</u> |
|------------|---------------|--------------|
| 1          | Science       | Cricket      |
| 1          | Maths         | Hockey       |
| 2          | DBMS          | Cricket      |
| 2          | PHP           | Hockey       |

| <u>SID</u> | <u>Course</u> |
|------------|---------------|
| 1          | Science       |
| 1          | Maths         |
| 2          | DBMS          |
| 2          | PHP           |

| <u>SID</u> | <u>Hobby</u> |
|------------|--------------|
| 1          | Cricket      |
| 1          | Hockey       |
| 2          | Cricket      |
| 2          | Hockey       |

Fig: Decomposing the relation into two 4NF relations

## 4NF: Example

- ◆ Example:

**Emp**

| <u>Ename</u> | <u>Pname</u> | <u>Dependent_name</u> |
|--------------|--------------|-----------------------|
| Ram          | X            | Rakesh                |
| Ram          | Y            | Rao                   |
| Ram          | X            | Rao                   |
| Ram          | Y            | Rakesh                |

**Emp\_Projects**

| <u>Ename</u> | <u>Pname</u> |
|--------------|--------------|
| Ram          | X            |
| Ram          | Y            |

**Emp\_Dependents**

| <u>Ename</u> | <u>Dependent_name</u> |
|--------------|-----------------------|
| Ram          | Rakesh                |
| Ram          | Rao                   |

**Fig: Decomposing the Emp relation into two 4NF relations**

# Properties of Relational Decompositions

- ◆ **Relation Decomposition and Insufficiency of Normal Forms**
- ◆ The relational database design algorithms that we present in start from a single **universal relation schema**  $R = \{A_1, A_2, \dots, A_n\}$  that includes *all* the attributes of the database. We implicitly make the **universal relation assumption**, which states that every attribute name is unique. The set  $F$  of functional dependencies that should hold on the attributes of  $R$  is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema  $R$  into a set of relation schemas  $D = \{R_1, R_2, \dots, R_m\}$  that will become the relational database schema;  $D$  is called a **decomposition** of  $R$ .

# Properties of Relational Decompositions

- ◆ **Relation Decomposition and Insufficiency of Normal Forms**
- ◆ We must make sure that each attribute in  $R$  will appear in at least one relation schema  $R_i$  in the decomposition so that no attributes are *lost*; formally, we have

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

- ◆ This is called the **attribute preservation** condition of a decomposition.

# Properties of Relational Decompositions

- ◆ **Relation Decomposition and Insufficiency of Normal Forms**
- ◆ Another goal is to have each individual relation  $R_i$  in the decomposition  $D$  be in BCNF or 3NF. However, this condition is not sufficient to guarantee a good database design on its own. We must consider the decomposition of the universal relation as a whole, in addition to looking at the individual relations.

# Properties of Relational Decompositions

- ◆ **Dependency Preservation Property of a Decomposition**
- ◆ It would be useful if each functional dependency  $X \rightarrow Y$  specified in  $F$  either appeared directly in one of the relation schemas  $Ri$  in the decomposition  $D$  or could be inferred from the dependencies that appear in some  $Ri$ . Informally, this is the *dependency preservation condition*. We want to preserve the dependencies because each dependency in  $F$  represents a constraint on the database. If one of the dependencies is not represented in some individual relation  $Ri$  of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.

# Properties of Relational Decompositions

- ◆ **Dependency Preservation Property of a Decomposition**
- ◆ It is not necessary that the exact dependencies specified in  $F$  appear themselves in individual relations of the decomposition  $D$ . It is sufficient that the union of the dependencies that hold on the individual relations in  $D$  be equivalent to  $F$ . We now define these concepts more formally.

# Properties of Relational Decompositions

- ◆ **Dependency Preservation Property of a Decomposition**
- ◆ Given a set of dependencies  $F$  on  $R$ , the **projection** of  $F$  on  $Ri$ , denoted by  $\pi_{Ri}(F)$  where  $Ri$  is a subset of  $R$ , is the set of dependencies  $X \rightarrow Y$  in  $F^+$  such that the attributes in  $X \cup Y$  are all contained in  $Ri$ . Hence, the projection of  $F$  on each relation schema  $Ri$  in the decomposition  $D$  is the set of functional dependencies in  $F^+$ , the closure of  $F$ , such that all the left- and right-hand-side attributes of those dependencies are in  $Ri$ . We say that a decomposition  $D = \{R1, R2, \dots, Rm\}$  of  $R$  is **dependency-preserving** with respect to  $F$  if the union of the projections of  $F$  on each  $Ri$  in  $D$  is equivalent to  $F$ ; that is,  $((\pi_{R1}(F)) \cup K \cup (\pi_{Rm}(F)))^+ = F^+$ .

# Properties of Relational Decompositions

- ◆ **Dependency Preservation Property of a Decomposition**
- ◆ If a decomposition is not dependency-preserving, some dependency is **lost** in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all left- and right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN—an option that is not practical.

# Properties of Relational Decompositions

- ◆ **Nonadditive (Lossless) Join Property of a Decomposition**
- ◆ Another property that a decomposition  $D$  should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition.
- ◆ Because this is a property of a decomposition of relation *schemas*, the condition of no spurious tuples should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in  $F$ . Hence, the lossless join property is always defined with respect to a specific set  $F$  of dependencies.

# Properties of Relational Decompositions

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

# Properties of Relational Decompositions

- ◆ **Nonadditive (Lossless) Join Property of a Decomposition**
- ◆ The word *loss* in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT ( $\pi$ ) and NATURAL JOIN (\*) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term *nonadditive join* because it describes the situation more accurately.
- ◆ The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term **lossy design** to refer to a design that represents a loss of information.

# **Examples of Normalization**

# 1NF: Example

| MP_ID | EMP_NAME | EMP_PHONE                 | EMP_STATE |
|-------|----------|---------------------------|-----------|
| 14    | John     | 7272826385,<br>9064738238 | UP        |
| 20    | Harry    | 8574783832                | Bihar     |
| 12    | Sam      | 7390372389,<br>8589830302 | Punjab    |

# 1NF: Solution Example

| EMP_ID | EMP_NAME | EMP_STATE |
|--------|----------|-----------|
| 14     | John     | UP        |
| 20     | Harry    | Bihar     |
| 12     | Sam      | Punjab    |

| EMP_ID | EMP_PHONE  |
|--------|------------|
| 14     | 7272826385 |
| 14     | 9064738238 |
| 20     | 8574783832 |
| 12     | 7390372389 |
| 12     | 8589830302 |

# 2NF: Example

| <u>Employee No</u> | <u>Department No</u> | Employee Name | Department |
|--------------------|----------------------|---------------|------------|
| 1                  | 101                  | Amit          | OBIEE      |
| 2                  | 102                  | Divya         | COGNOS     |
| 3                  | 101                  | Rama          | OBIEE      |

Given FD:

{Employee NO, Department No}  $\rightarrow$  Employee Name  
Department No  $\rightarrow$  Department

# 2NF: Solution Example

| <u>Employee No</u> | <u>Department No</u> | Employee Name |
|--------------------|----------------------|---------------|
| 1                  | 101                  | Amit          |
| 2                  | 102                  | Divya         |
| 3                  | 101                  | Rama          |

| <u>Department No</u> | Department |
|----------------------|------------|
| 101                  | OBIEE      |
| 102                  | COGNOS     |

# 3NF: Example

| <u>Employee No</u> | Salary Slip No | Employee Name | Salary |
|--------------------|----------------|---------------|--------|
| 1                  | 0001           | Amit          | 50000  |
| 2                  | 0002           | Divya         | 40000  |
| 3                  | 0003           | Rama          | 57000  |

Given FD:

Employee No-> Employee Name, Salary Slip No

Salary Slip No -> Salary

# 3NF: Solution Example

| <u>Employee No</u> | Salary Slip No | Employee Name |
|--------------------|----------------|---------------|
| 1                  | 0001           | Amit          |
| 2                  | 0002           | Divya         |
| 3                  | 0003           | Rama          |

| <u>Salary Slip No</u> | Salary |
|-----------------------|--------|
| 0001                  | 50000  |
| 0002                  | 40000  |
| 0003                  | 57000  |

# 4NF: Example

| Pizza Delivery Permutations |                      |                      |
|-----------------------------|----------------------|----------------------|
| <u>Restaurant</u>           | <u>Pizza Variety</u> | <u>Delivery Area</u> |
| A1 Pizza                    | Thick Crust          | Springfield          |
| A1 Pizza                    | Thick Crust          | Shelbyville          |
| A1 Pizza                    | Thick Crust          | Capital City         |
| A1 Pizza                    | Stuffed Crust        | Springfield          |
| A1 Pizza                    | Stuffed Crust        | Shelbyville          |
| A1 Pizza                    | Stuffed Crust        | Capital City         |
| Elite Pizza                 | Thin Crust           | Capital City         |
| Elite Pizza                 | Stuffed Crust        | Capital City         |
| Vincenzo's Pizza            | Thick Crust          | Springfield          |
| Vincenzo's Pizza            | Thick Crust          | Shelbyville          |
| Vincenzo's Pizza            | Thin Crust           | Springfield          |
| Vincenzo's Pizza            | Thin Crust           | Shelbyville          |

# 4NF: Example

- ◆ The dependencies are:

{Restaurant} →→ {Pizza Variety}

{Restaurant} →→ {Delivery Area}

# 4NF: Solution Example

| <u>Restaurant</u> | <u>Pizza Variety</u> |
|-------------------|----------------------|
| A1 Pizza          | Thick Crust          |
| A1 Pizza          | Stuffed Crust        |
| Elite Pizza       | Thin Crust           |
| Elite Pizza       | Stuffed Crust        |
| Vincenzo's Pizza  | Thick Crust          |
| Vincenzo's Pizza  | Thin Crust           |

| <u>Restaurant</u> | <u>Delivery Area</u> |
|-------------------|----------------------|
| A1 Pizza          | Springfield          |
| A1 Pizza          | Shelbyville          |
| A1 Pizza          | Capital City         |
| Elite Pizza       | Capital City         |
| Vincenzo's Pizza  | Springfield          |
| Vincenzo's Pizza  | Shelbyville          |

# Normalization Example : 1NF

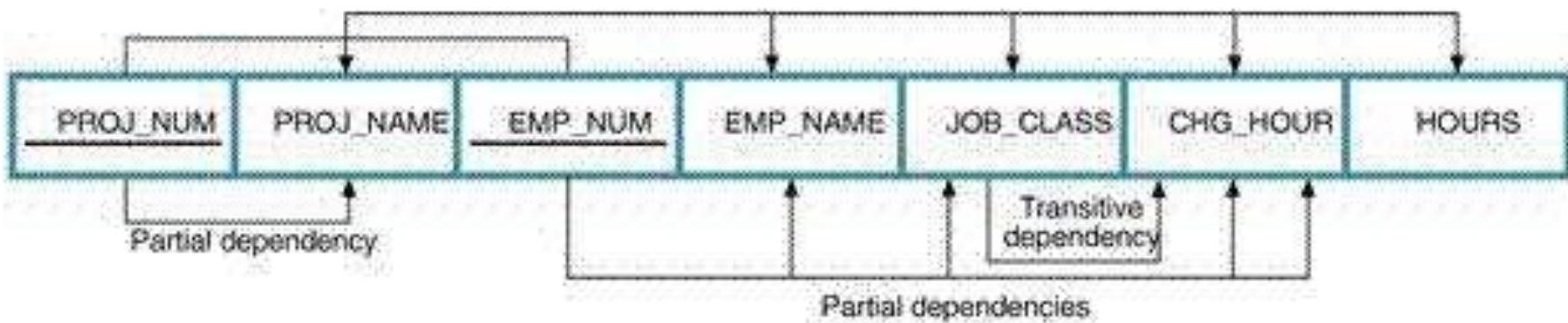


FIGURE 5.4 A DEPENDENCY DIAGRAM: FIRST NORMAL FORM (1NF)

# Normalization Example: 2NF Results

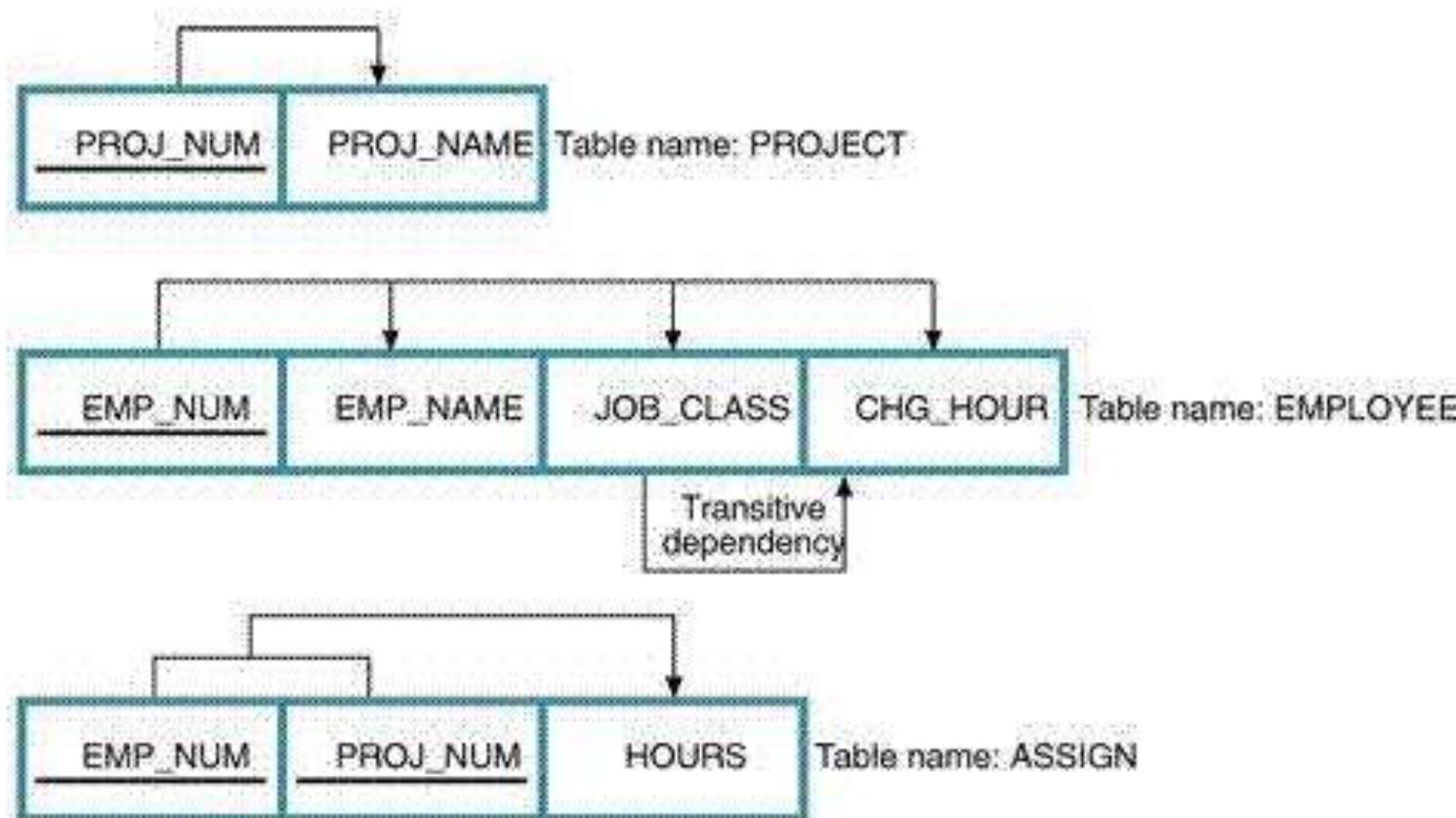


FIGURE 5.5 ■ SECOND NORMAL FORM (2NF) CONVERSION RESULTS

# Normalization Example: 3NF Results

Project (Proj\_Num, Proj\_name)

Employee( Emp\_Num, Emp\_name, Job\_Class)

Job(Job\_Class, CHG\_Hour)

Assign(Emp\_Num, Proj\_Num, Hours)

# **Database Management System (MDS 505)**

## **Jagdish Bhatta**

# Unit – 4

# SQL & Query Optimization

# **SQL Standards ,Data types**

- ◆ We have already covered these in unit 2.

# **Database Objects- DDL-DML-DCL-TCL**

- ◆ We have already covered these in unit 2

# Embedded SQL

- ♦ **Embedded SQL** is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL. Embedded SQL statements are SQL statements written inline with the program source code, of the host language. The embedded SQL statements are parsed by an embedded SQL preprocessor and replaced by host-language calls to a code library. The output from the preprocessor is then compiled by the host compiler. This allows programmers to embed SQL statements in programs written in any number of languages such as C/C++. E.g.: Proc\*C is Oracle's embedded SQL environment.

# Embedded SQL

- ◆ E.g: Embedding SQL into C.

```
{
 int a;
 /* ... */
 EXEC SQL SELECT salary INTO :a
 FROM Employee
 WHERE SSN=876543210;
 /* ... */
 printf("The salary is %d\n", a);
 /* ... */
}
```

All SQL statements need to start with EXEC SQL and end with a semicolon ";". You can place the SQL statements anywhere within a C/C++ block, with the restriction that the declarative statements do not come after the executable statements.

# Static Vs Dynamic SQL

- ♦ In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, anytime we want to write a different query, we must modify the program code and go through all the steps involved (compiling, debugging, testing, and so on). In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*. For example, we may want to write a program that accepts an SQLquery typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs. Another example is when a user-friendly interface generates SQL queries dynamically for the user based on user input through a Web interface or mobile App. **Dynamic SQL**, which is one technique for writing this type of database program.

# Query Processing

- ◆ **Query processing refers to the range of activities involved in extracting data from a database.** The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.
- ◆ The basic steps are:
  - **Parsing and translation.**
  - **Optimization.**
  - **Evaluation.**

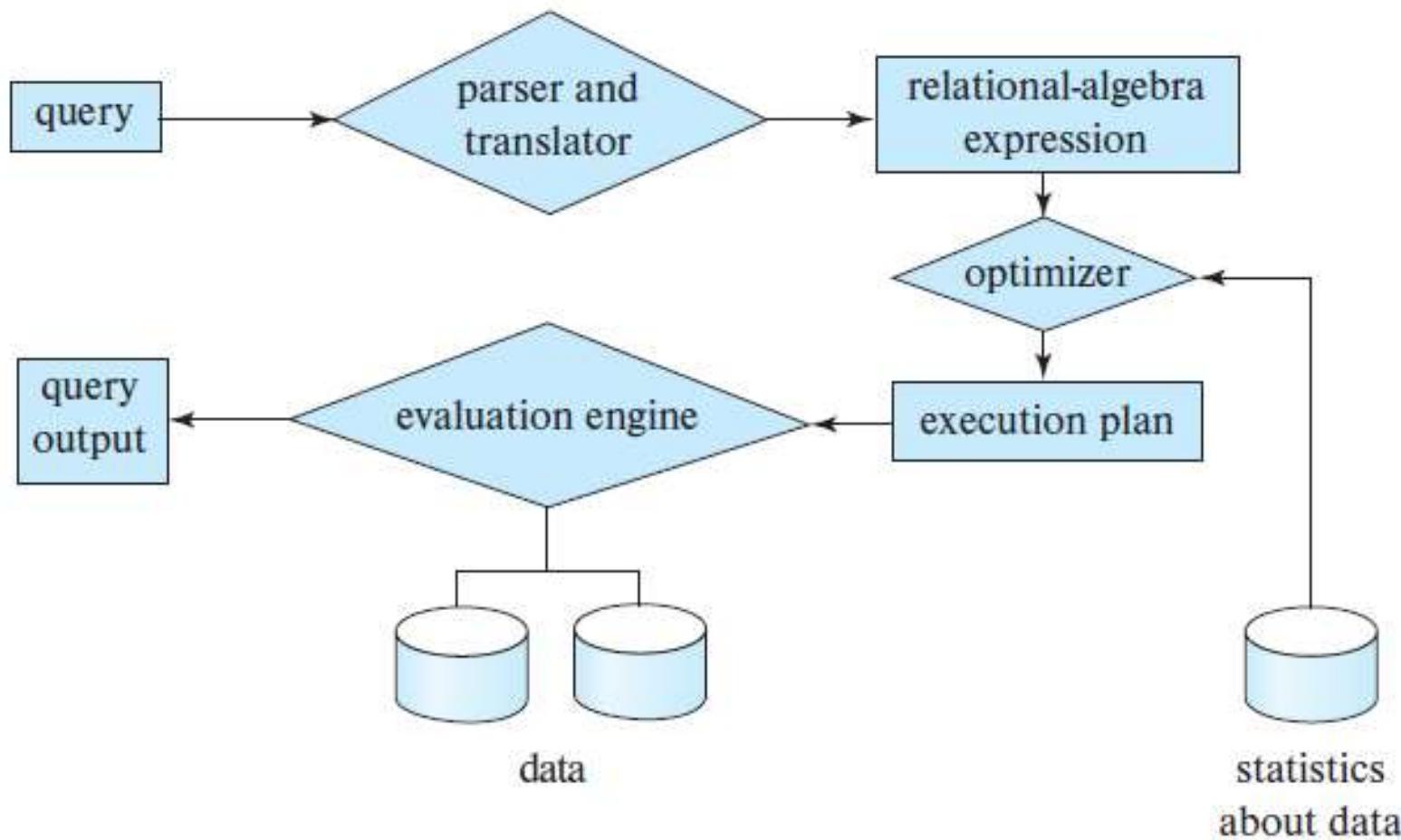
# Query Processing

- ◆ Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

# Query Processing

- ◆ Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

# Query Processing



**Figure 15.1** Steps in query processing.

# Query Processing

- Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relational algebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions.

# Query Processing

- ◆ Consider the query;

**select *salary***

**from *instructor***

**where *salary* < 75000;**

- ◆ This query can be translated into either of the following relational-algebra expressions:

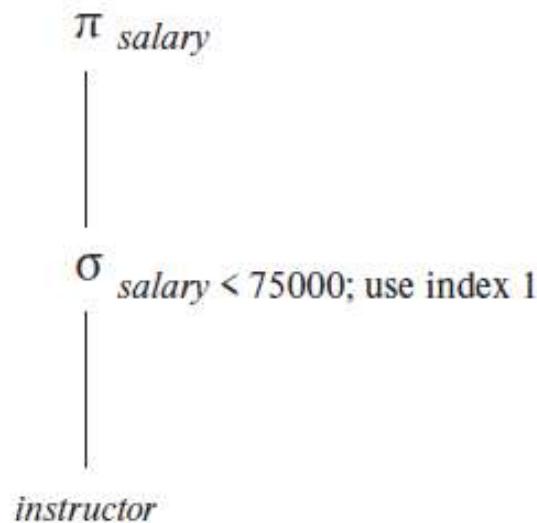
$$\sigma_{\text{salary} < 75000} (\Pi_{\text{salary}} (\text{instructor}))$$

$$\Pi_{\text{salary}} (\sigma_{\text{salary} < 75000} (\text{instructor}))$$

# Query Processing

- ◆ Further, we can execute each relational-algebra operation by one of several different algorithms.
- ◆ To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use.
- ◆ A relational algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**. A **sequence of primitive operations that can be used to evaluate a query** is a **query-execution plan or query-evaluation plan**.
- ◆ The **query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query**.

# Query Processing



**Figure 15.2** A query-evaluation plan.

# Query Processing

- ◆ The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation; this task is called *query optimization*.
- ◆ Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.
- ◆ The sequence of steps already described for processing a query is representative; not all databases exactly follow those steps. For instance, instead of using the relational-algebra representation, several databases use an annotated parse tree representation based on the structure of the given SQL query

# Query Processing

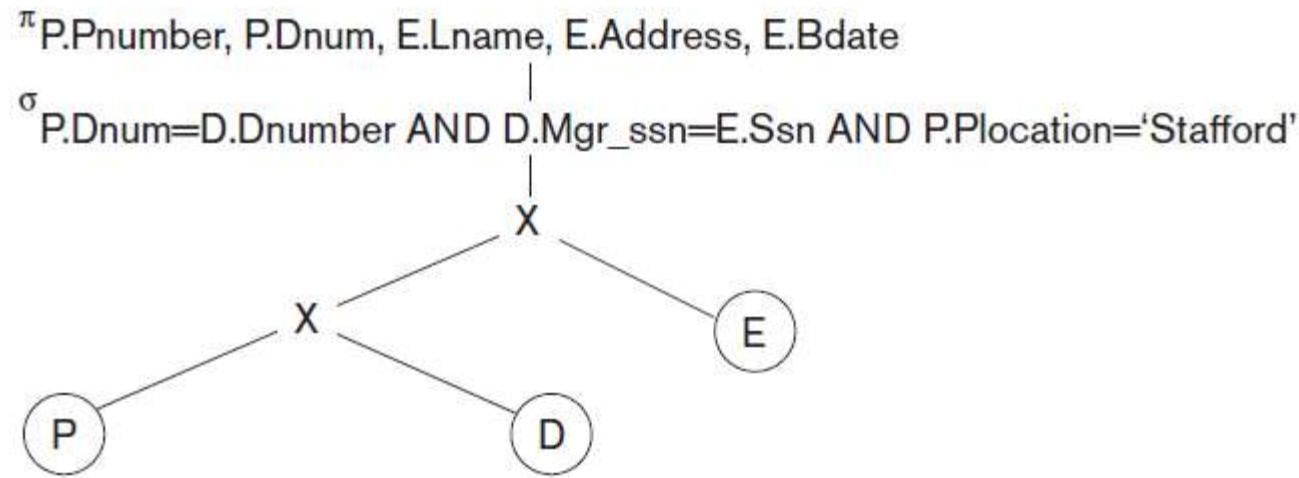
- ◆ In order to optimize a query, a query optimizer must know the cost of each operation. Although the exact cost is hard to compute, since it depends on many parameters such as actual memory available to the operation, it is possible to get a rough estimate of execution cost for each operation.

# Query Tree

- ◆ A **query tree** is a tree data structure that corresponds to an extended relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and it represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query

# Query Tree

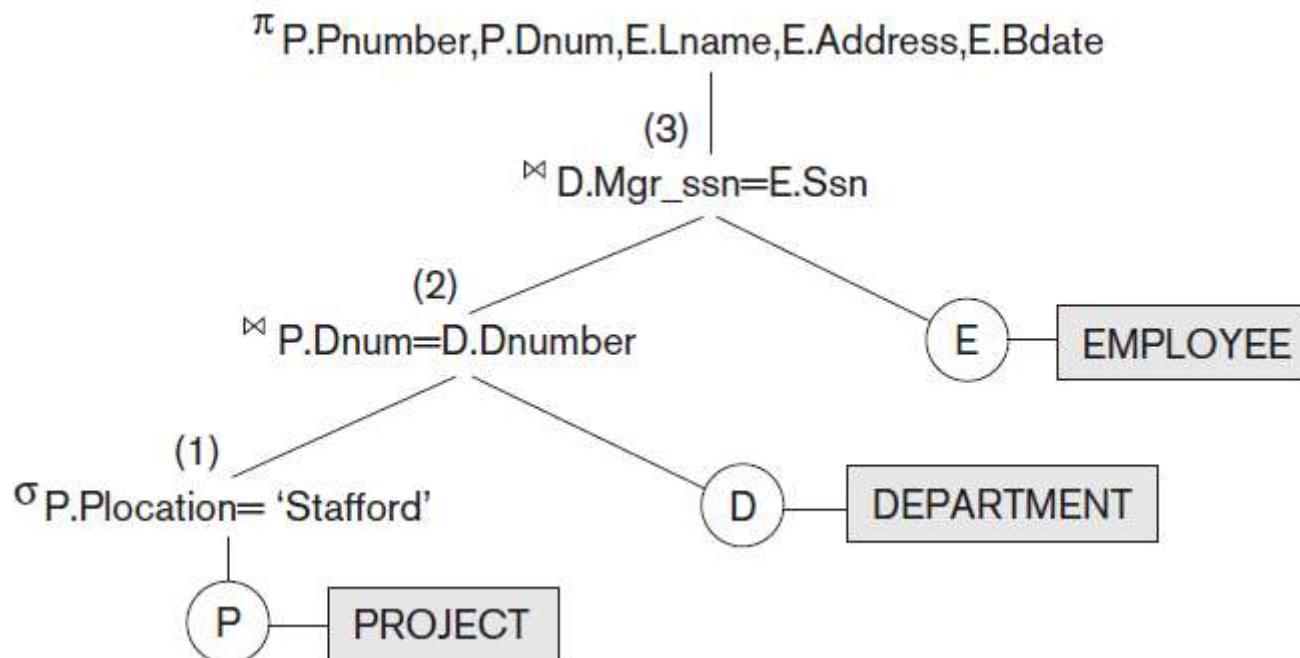
◆ **SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate  
FROM PROJECT P, DEPARTMENT D, EMPLOYEE E  
WHERE P.Dnum=D.Dnumber AND D.Mgr\_ssn=E.Ssn AND  
P.Plocation='Stafford';**



# Query Tree

◆ **SELECT** P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate  
**FROM** PROJECT P, DEPARTMENT D, EMPLOYEE E  
**WHERE** P.Dnum=D.Dnumber **AND** D.Mgr\_ssn=E.Ssn **AND**  
P.Plocation='Stafford';

$\pi_{Pnumber, Dnum, Lname, Address, Bdate} (((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT) \bowtie_{Mgr\_ssn=Ssn}(EMPLOYEE))$



# Measures of Query Cost

- ◆ There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan.
- ◆ The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication.
- ◆ In large database systems, the cost to access data from disk is usually the most important cost, since disk accesses are slow compared to in-memory operations.

# Measures of Query Cost

- ◆ The *number of block transfers from disk and the number of disk seeks* can be used to estimate the cost of a query-evaluation plan. If the disk subsystem takes an average of  $tT$  *seconds to transfer a block of data, and has an average block-access time* (disk seek time plus rotational latency) of  $tS$  *seconds, then an operation that transfers b blocks and performs S seeks would take  $b * tT + S * tS$  seconds.*
- ◆ *The values of  $tT$  and  $tS$  must be calibrated for the disk system used, but typical values for high-end disks today would be  $tS = 4$  milliseconds and  $tT = 0.1$  milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.*
- ◆ One can refine the cost estimates further by distinguishing block reads from block writes, since block writes are typically about twice as expensive as reads (this is because disk systems read sectors back after they are written to verify that the write was successful).

# Measures of Query Cost

- ◆ The **response time for a query-evaluation plan** (that is, the **wall-clock time** required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan.
- ◆ Unfortunately, the response time of a plan is very hard to estimate without actually executing the plan, for the following reasons:
  - **The response time depends on the contents of the buffer when the query begins execution;** this information is not available when the query is optimized, and is hard to account for even if it were available.
  - **In a system with multiple disks, the response time depends on how accesses are distributed among disks,** which is hard to estimate without detailed knowledge of data layout on disk.

# Measures of Query Cost

- ◆ Optimizers often try to minimize the total **resource** consumption of a query plan. Where resource consumption may be extra disk reads, parallel reads across multiple disks etc.

# Evaluation of Expressions

- ◆ The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk. An alternative approach is to evaluate several operations simultaneously in a **pipeline**, with **the results of one operation passed on to the next**, without the need to store a temporary relation.

# Evaluation of Expressions

- ◆ Materialization:

- ◆ Consider the expression;

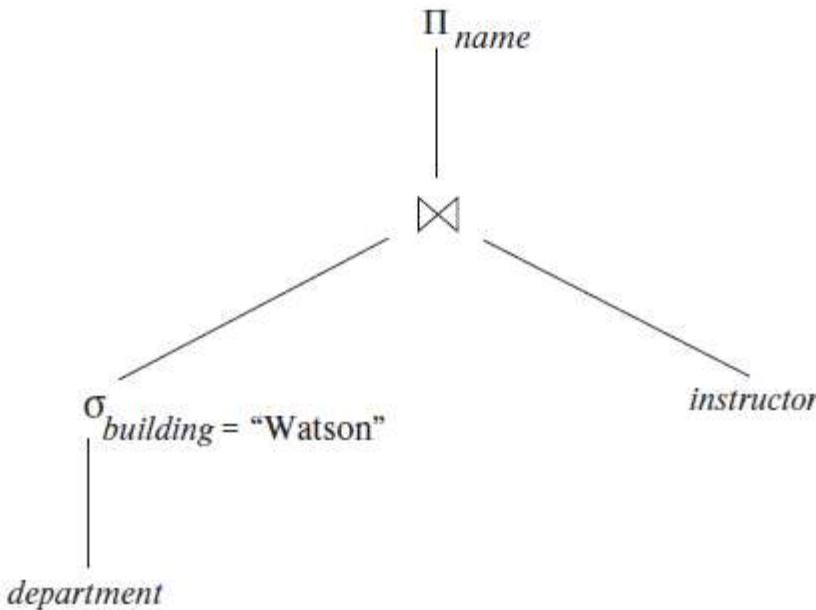
$$\Pi_{name}(\sigma_{building = "Watson"}(department) \bowtie instructor)$$


Figure 15.11 Pictorial representation of an expression.

# Evaluation of Expressions

- ◆ **Materialization:**
- ◆ If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation: *the selection operation on department*. The inputs to the lowest-level operations are relations in the database. We execute these operations by the algorithms that we studied earlier, like selection, join etc, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In our example, the inputs to the join are the *instructor* relation and the temporary relation created by the *selection on department*. The join can now be evaluated, creating another temporary relation.

# Evaluation of Expressions

- ◆ **Materialization:**
- ◆ By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In the earlier example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.
- ◆ This is known as **materialized evaluation**, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.
- ◆ To compute the cost of evaluating an expression as done here, we have to add the costs of all the operations, as well as the cost of writing the intermediate results to disk.

# Evaluation of Expressions

- ◆ **Pipelining:**
- ◆ We can improve query-evaluation efficiency by reducing the number of temporary files that are produced. We achieve this reduction by combining several relational operations into a *pipeline of operations*, *in which the results of one operation* are passed along to the next operation in the pipeline. This is called **pipelined evaluation**.
- ◆ For example, consider the expression  $(\Pi_{a1,a2}(r \bowtie s))$ . *If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection.* These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result, and instead create the final result directly.

# Evaluation of Expressions

- ◆ **Pipelining:**
- ◆ Creating a pipeline of operations can provide two benefits:
  - It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.
  - It can start generating query results quickly, if the root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

# Query Optimization

- ◆ **Query optimization is the process of selecting the most efficient query-evaluation plan** from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play.
- ◆ Query optimization is the process of executing a SQL query in relational databases to determine the most efficient way to execute a given query by considering the possible query plans. The goal of query optimization is to optimize the given query for the sake of efficiency.

# Query Optimization

- ◆ Query optimization in database has gained significant importance as it helps to reduce the size, memory usage and time required for any query to be processed.
- ◆ The main objective of any query optimization is to determine the best strategy for executing each query.

# Query Optimization

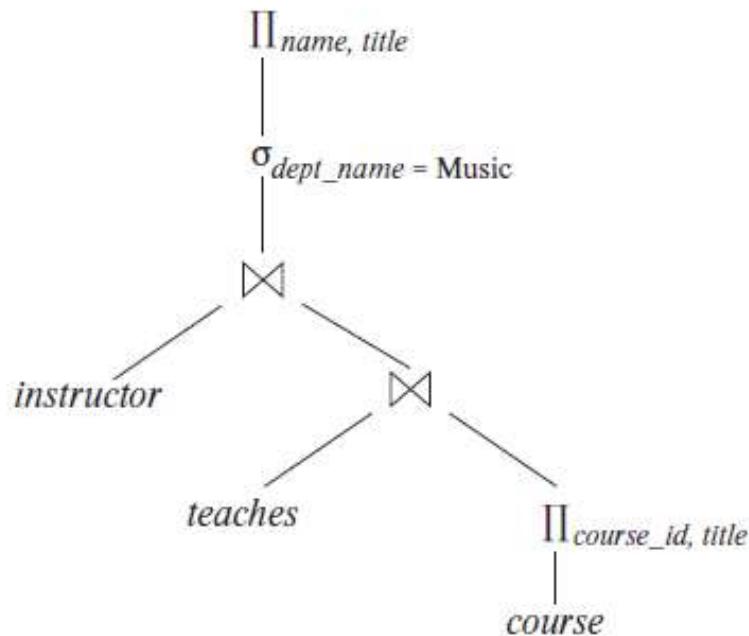
- ◆ Consider the following relational-algebra expression, for the query “Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach.”

$$\Pi_{name, title} (\sigma_{dept\_name = "Music"} (instructor \bowtie (teaches \bowtie \Pi_{course\_id, title}(course))))$$

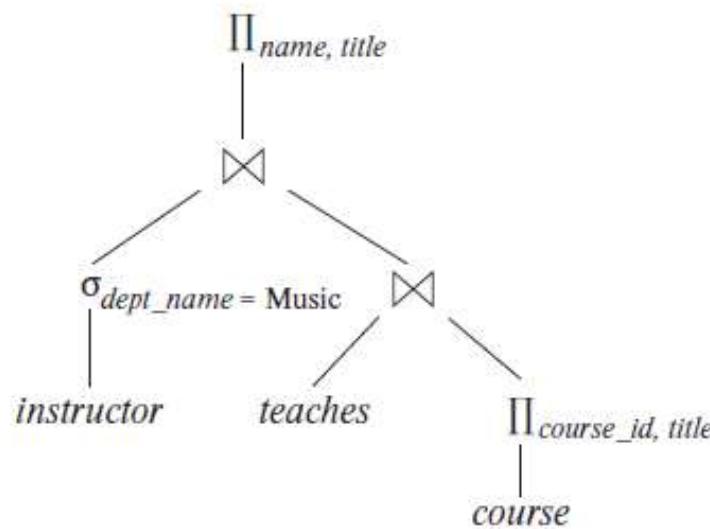
- ◆ Since we are concerned with only those tuples in the *instructor relation that pertain to the Music department*, we do not need to consider those tuples that do not have *dept name = “Music”*. By reducing the number of tuples of the *instructor relation that we need to access*, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression:

$$\Pi_{name, title} ((\sigma_{dept\_name = "Music"} (instructor)) \bowtie (teaches \bowtie \Pi_{course\_id, title}(course)))$$

# Query Optimization



(a) Initial expression tree



(b) Transformed expression tree

**Figure 16.1** Equivalent expressions.

# Query Optimization

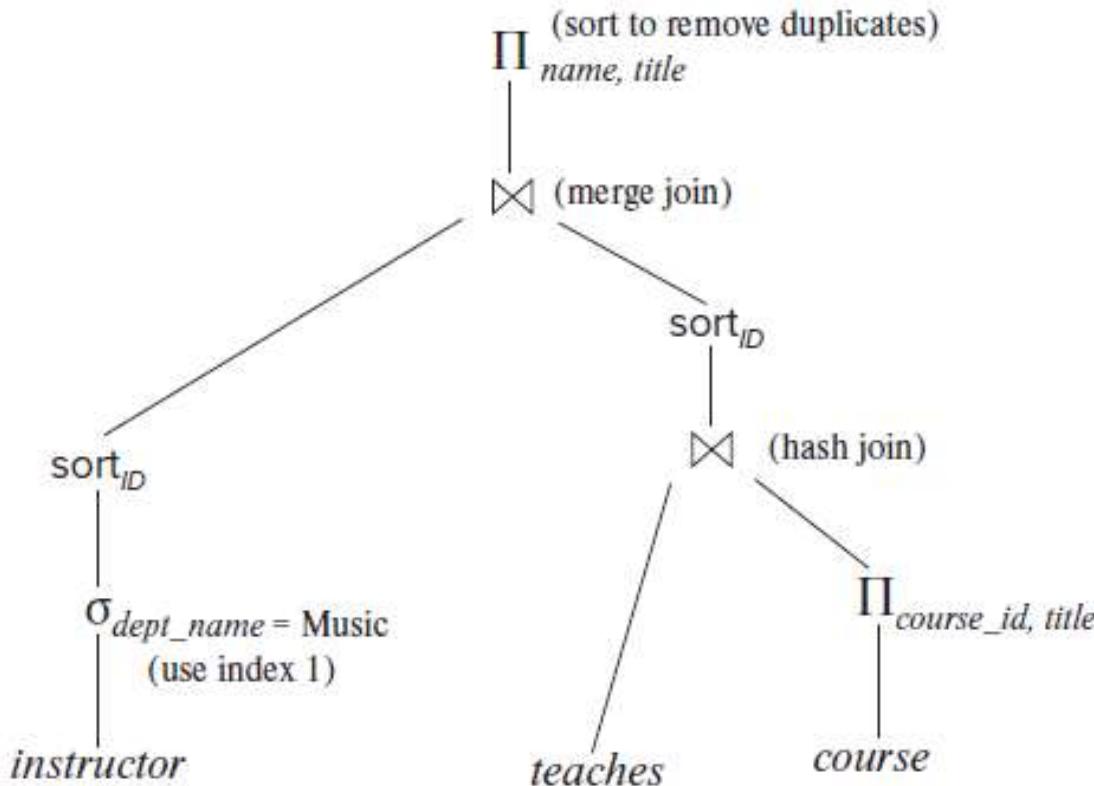


Figure 16.2 An evaluation plan.

# Query Optimization

- ◆ An evaluation plan defines exactly what algorithm should be used for each operation and how the execution of the operations should be coordinated. Figure 16.2 illustrates one possible evaluation plan for the expression from Figure 16.1(b).
- ◆ As we have seen, several different algorithms can be used for each relational operation, giving rise to alternative evaluation plans. In the figure, hash join has been chosen for one of the join operations, while the other uses merge join, after sorting the relations on the join attribute, which is ID. All edges are assumed to be pipelined, unless marked as materialized. With pipelined edges the output of the producer is sent directly to the consumer, without being written out to disk.

# Query Optimization

- ◆ Given a relational-algebra expression, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least costly way of generating the result (or, at least, is not much costlier than the least costly way).

# Query Optimization

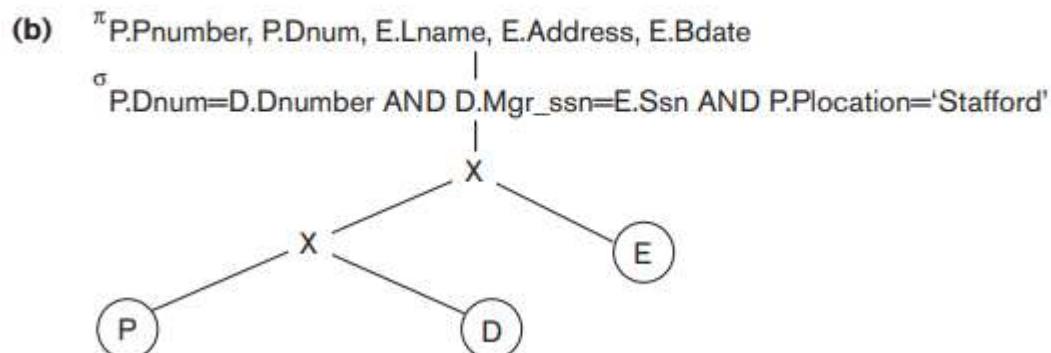
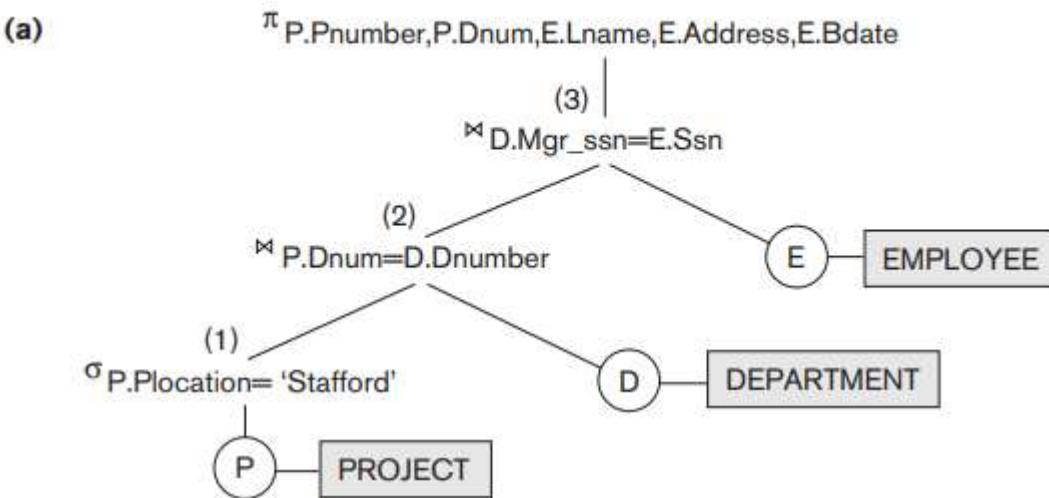
- ◆ To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one. Generation of query-evaluation plans involves three steps:
  - generating expressions that are logically equivalent to the given expression,
  - annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans, and
  - estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least

# Heuristic Optimization of Query Trees

- ◆ Many different query trees—can be semantically equivalent; that is, they can represent the same query and produce the same results.
- ◆ The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization. For example, for a SELECTPROJECT-JOIN query, in next slide, the initial tree is shown in Figure (b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is very inefficient if executed directly, because of the CARTESIAN PRODUCT ( $\times$ ) operations.

# Heuristic Optimization of Query Trees

```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
P.Plocation='Stafford';
```



# Heuristic Optimization of Query Trees

- ♦ For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each.
- ♦ However, this canonical query tree in Figure b is in a simple standard form that can be easily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent final query tree that is efficient to execute.

# Heuristic Optimization of Query Trees

- ◆ The optimizer must include rules for equivalence among extended relational algebra expressions that can be applied to transform the initial tree into the final, optimized query tree.
- ◆ Consider the following query Q on the database: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’. This query can be specified in SQL as follows:

Q: SELECT E.Lname

```
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE P.Pname='Aquarius' AND P.Pnumber = W.Pno
AND E.Essn=W.Ssn AND E.Bdate > '1957-12-31';
```

# Heuristic Optimization of Query Trees

- ◆ The initial query tree for Q is shown in Figure 19.2(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS\_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to execute. This particular query needs only one record from the PROJECT relation—for the ‘Aquarius’ project—and only the EMPLOYEE records for those whose date of birth is after ‘1957-12-31’.
- ◆ Figure 19.2(b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

# Heuristic Optimization of Query Trees

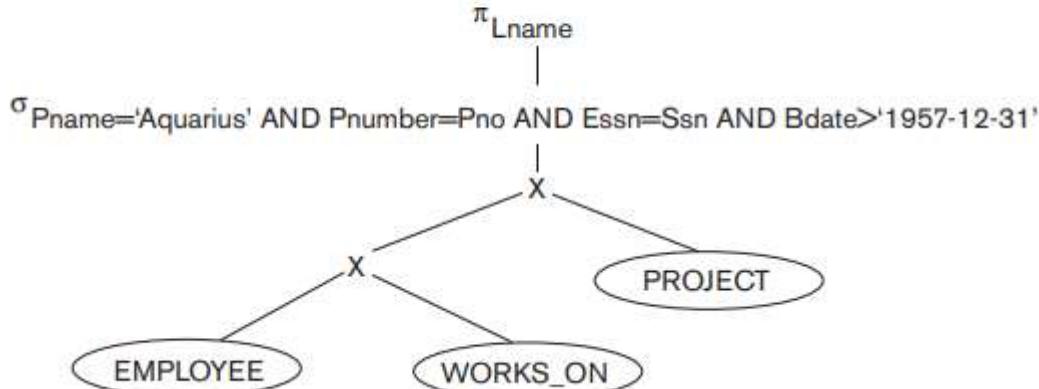
- ♦ A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 19.2(c). This uses the information that Pnumber is a key attribute and Pname is unique of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition as a selection with a JOIN operation, as shown in Figure 19.2(d). Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT ( $\pi$ ) operations as early as possible in the query tree, as shown in Figure 19.2(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

# Heuristic Optimization of Query Trees

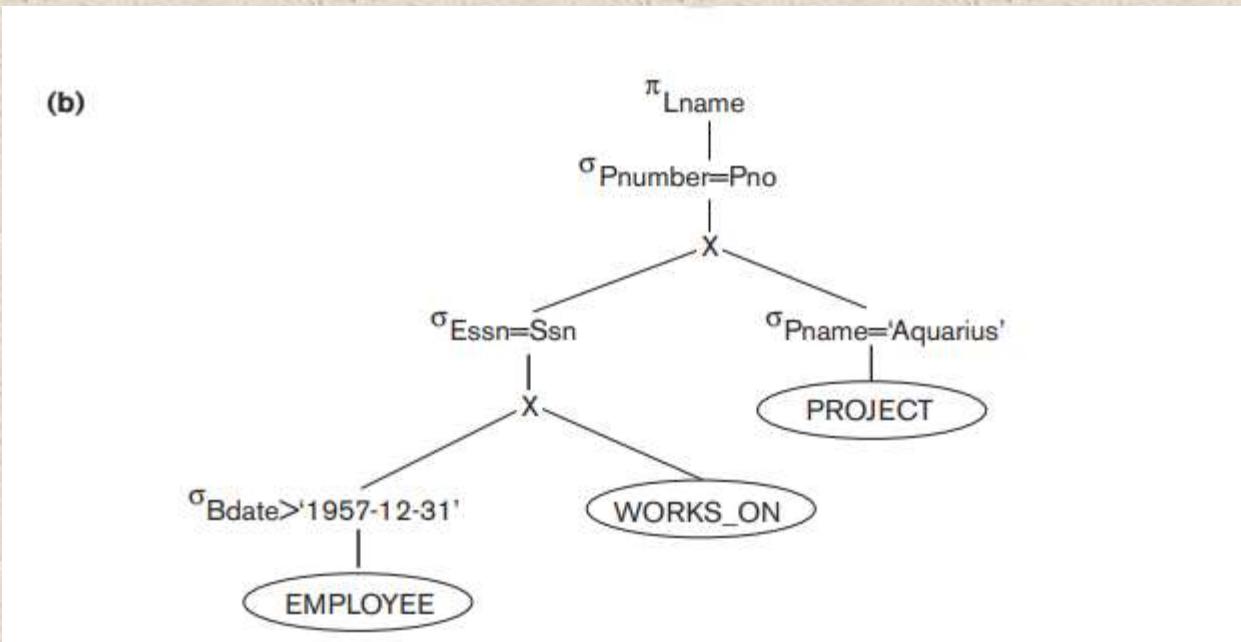
**Figure 19.2**

Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree. (c) Applying the more restrictive SELECT operation first.

(a)

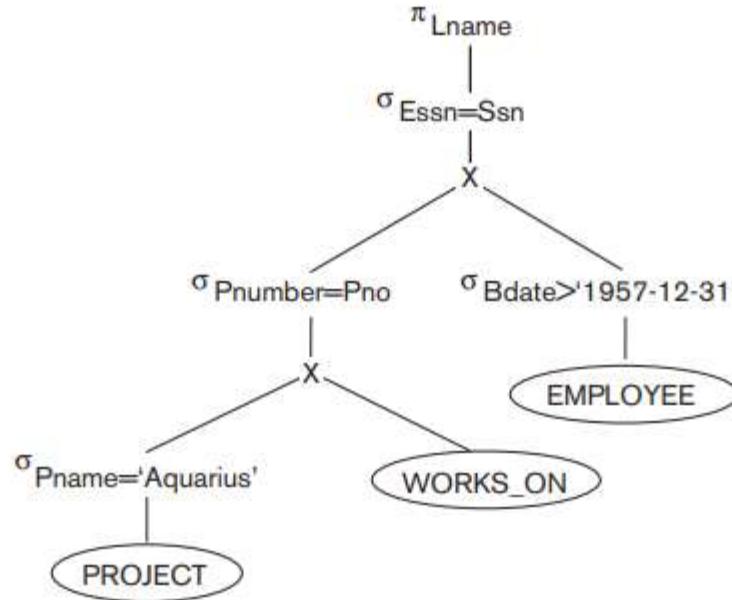


# Heuristic Optimization of Query Trees



# Heuristic Optimization of Query Trees

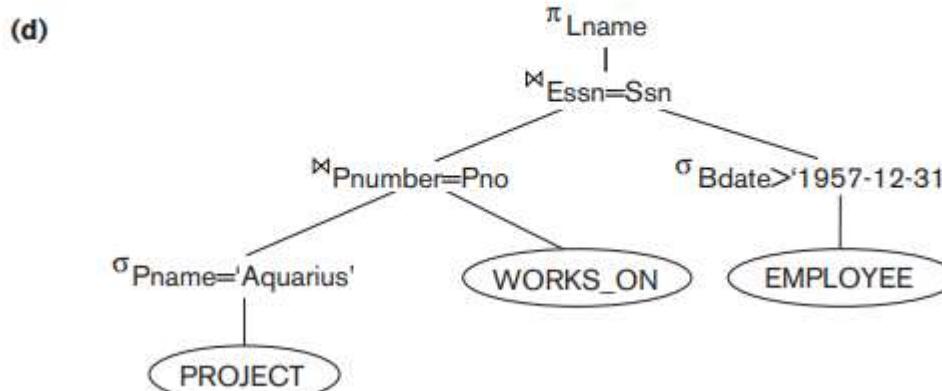
(c)



# Heuristic Optimization of Query Trees

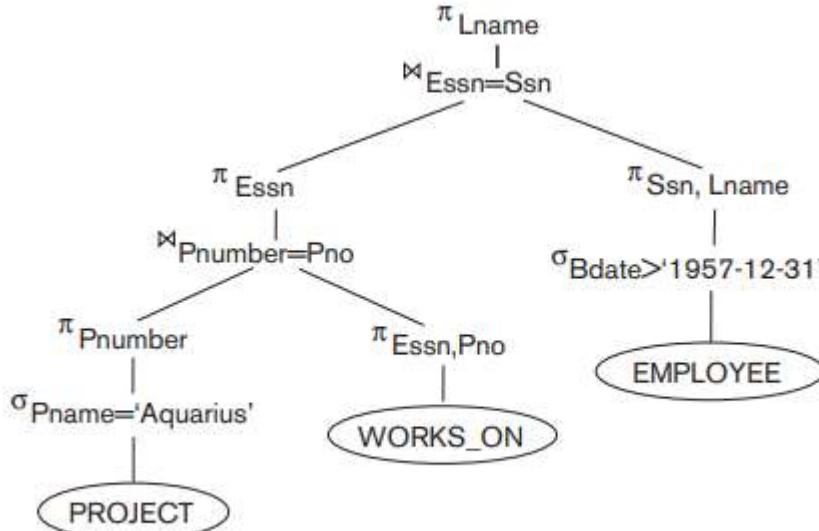
**Figure 19.2 (continued)**

Steps in converting a query tree during heuristic optimization. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations. (e) Moving PROJECT operations down the query tree.



# Heuristic Optimization of Query Trees

(e)



# Heuristic Optimization of Query Trees

- ◆ As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules preserve this equivalence.

# General Transformation Rules for Relational Algebra Operations

- ◆ There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a different order but the two relations represent the same information, we consider the relations to be equivalent.

# General Transformation Rules for Relational Algebra Operations

1. **Cascade of  $\sigma$ .** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual  $\sigma$  operations:

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

2. **Commutativity of  $\sigma$ .** The  $\sigma$  operation is commutative:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

3. **Cascade of  $\pi$ .** In a cascade (sequence) of  $\pi$  operations, all but the last one can be ignored:

$$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\dots(\pi_{\text{List}_n}(R))\dots)) \equiv \pi_{\text{List}_1}(R)$$

4. **Commuting  $\sigma$  with  $\pi$ .** If the selection condition  $c$  involves only those attributes  $A_1, \dots, A_n$  in the projection list, the two operations can be commuted:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

# General Transformation Rules for Relational Algebra Operations

5. **Commutativity of  $\bowtie$  (and  $\times$ ).** The join operation is commutative, as is the  $\times$  operation:

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

6. **Commuting  $\sigma$  with  $\bowtie$  (or  $\times$ ).** If all the attributes in the selection condition  $c$  involve only the attributes of one of the relations being joined—say,  $R$ —the two operations can be commuted as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

Alternatively, if the selection condition  $c$  can be written as  $(c_1 \text{ AND } c_2)$ , where condition  $c_1$  involves only the attributes of  $R$  and condition  $c_2$  involves only the attributes of  $S$ , the operations commute as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$$

The same rules apply if the  $\bowtie$  is replaced by a  $\times$  operation.

# General Transformation Rules for Relational Algebra Operations

7. **Commuting  $\pi$  with  $\bowtie$  (or  $\times$ ).** Suppose that the projection list is  $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$ , where  $A_1, \dots, A_n$  are attributes of  $R$  and  $B_1, \dots, B_m$  are attributes of  $S$ . If the join condition  $c$  involves only attributes in  $L$ , the two operations can be commuted as follows:

$$\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, \dots, A_n} (R)) \bowtie_c (\pi_{B_1, \dots, B_m} (S))$$

If the join condition  $c$  contains additional attributes not in  $L$ , these must be added to the projection list, and a final  $\pi$  operation is needed. For example, if attributes  $A_{n+1}, \dots, A_{n+k}$  of  $R$  and  $B_{m+1}, \dots, B_{m+p}$  of  $S$  are involved in the join condition  $c$  but are not in the projection list  $L$ , the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, \dots, A_n, A_{n+1}, \dots, A_{n+k}} (R)) \bowtie_c (\pi_{B_1, \dots, B_m, B_{m+1}, \dots, B_{m+p}} (S)))$$

For  $\times$ , there is no condition  $c$ , so the first transformation rule always applies by replacing  $\bowtie_c$  with  $\times$ .

8. **Commutativity of set operations.** The set operations  $\cup$  and  $\cap$  are commutative, but  $-$  is not.
9. **Associativity of  $\bowtie$ ,  $\times$ ,  $\cup$ , and  $\cap$ .** These four operations are individually associative; that is, if both occurrences of  $\theta$  stand for the same operation that is any one of these four operations (throughout the expression), we have:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

# General Transformation Rules for Relational Algebra Operations

10. **Commuting  $\sigma$  with set operations.** The  $\sigma$  operation commutes with  $\cup$ ,  $\cap$ , and  $-$ . If  $\theta$  stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c(R \theta S) \equiv (\sigma_c(R)) \theta (\sigma_c(S))$$

11. **The  $\pi$  operation commutes with  $\cup$ .**

$$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$$

12. **Converting a  $(\sigma, \times)$  sequence into  $\bowtie$ .** If the condition  $c$  of a  $\sigma$  that follows a  $\times$  corresponds to a join condition, convert the  $(\sigma, \times)$  sequence into a  $\bowtie$  as follows:

$$(\sigma_c(R \times S)) \equiv (R \bowtie_c S)$$

# General Transformation Rules for Relational Algebra Operations

## 13. Pushing $\sigma$ in conjunction with set difference.

$$\sigma_c(R - S) = \sigma_c(R) - \sigma_c(S)$$

However,  $\sigma$  may be applied to only one relation:

$$\sigma_c(R - S) = \sigma_c(R) - S$$

## 14. Pushing $\sigma$ to only one argument in $\cap$ .

If in the condition  $\sigma_c$  all attributes are from relation R, then:

$$\sigma_c(R \cap S) = \sigma_c(R) \cap S$$

## 15. Some trivial transformations.

If S is empty, then  $R \cup S = R$

If the condition c in  $\sigma_c$  is true for the entire R, then  $\sigma_c(R) = R$ .

# Outline of a Heuristic Algebraic Optimization Algorithm

- ◆ We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Step 1: Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
  - Step 2: Using Rules 2, 4, 6, and 10, 13, 14 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from only one table, which means that it represents a selection condition, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from two tables, which means that it represents a join condition, the condition is moved to a location down the tree after the two tables are combined

# Outline of a Heuristic Algebraic Optimization Algorithm

- ◆ We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Step 3: Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of most restrictive SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.
  - Step 4: Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.

# Outline of a Heuristic Algebraic Optimization Algorithm

- ◆ We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Step 5: Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
  - Step 6: Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

# Outline of a Heuristic Algebraic Optimization Algorithm

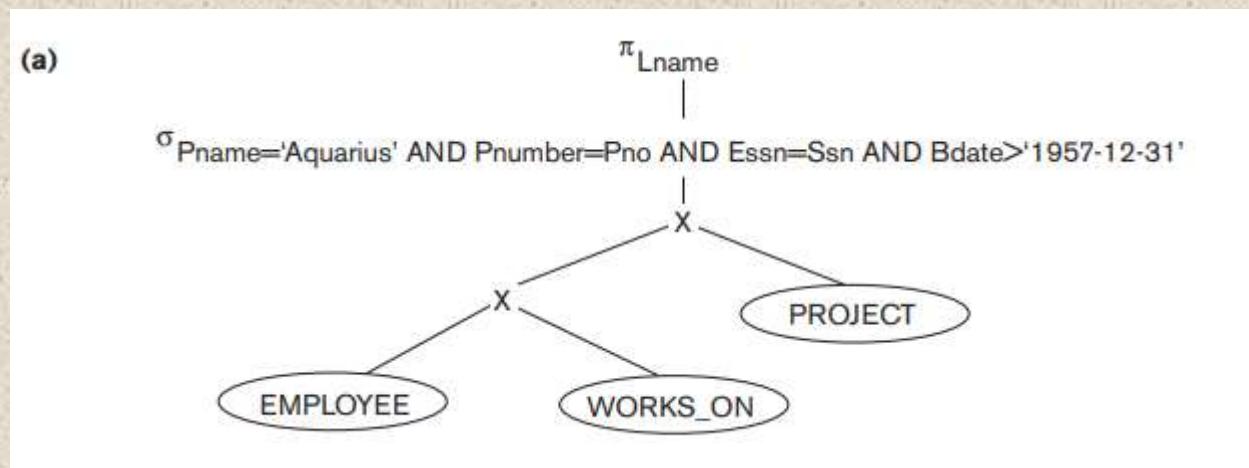
- ◆ We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Identify subtrees that represent groups of operations that can be executed by a single algorithm.

# Outline of a Heuristic Algebraic Optimization Algorithm

- ♦ In the example, Figure 19.2(b) shows the tree in Figure 19.2(a) after applying steps 1 and 2 of the algorithm;
- ♦ Figure 19.2(c) shows the tree after step 3; Figure 19.2(d) after step 4; and
- ♦ Figure 19.2(e) after step 5.
- ♦ In step 6, we may group together the operations in the subtree whose root is the operation  $\pi_{E_{\text{SSN}}}$  into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation  $\pi_{E_{\text{SSN}}}$ , because the first grouping means that this subtree is executed first.

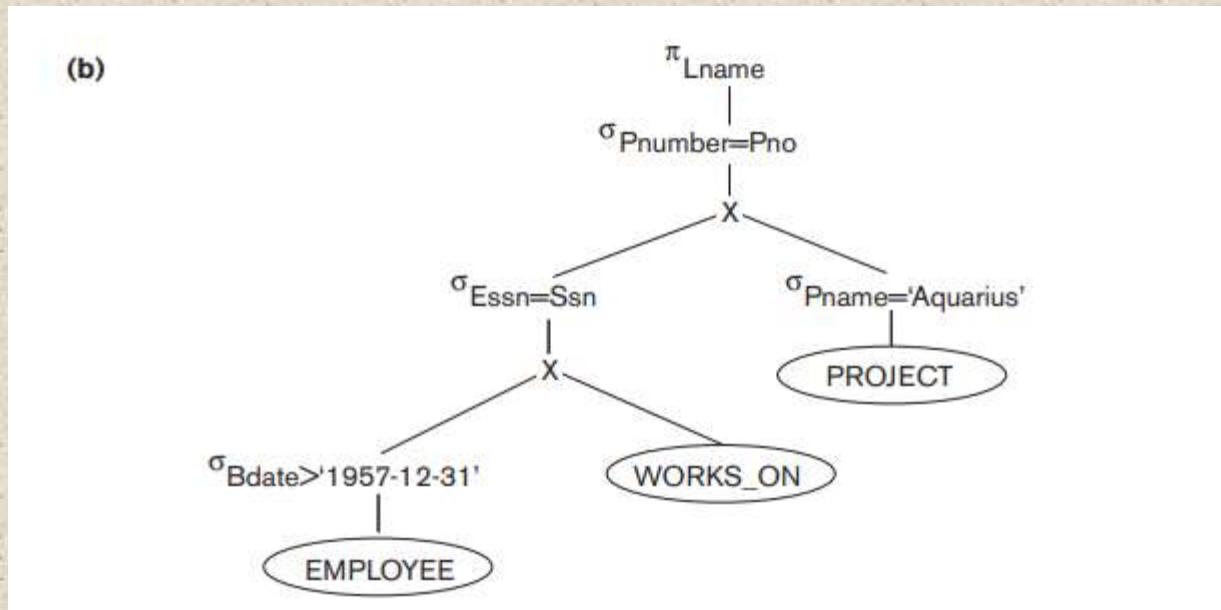
# Outline of a Heuristic Algebraic Optimization Algorithm

```
SELECT E.Lname
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE P.Pname='Aquarius' AND P.Pnumber = W.Pno AND
E.Essn=W.Ssn AND E.Bdate > '1957-12-31';
```



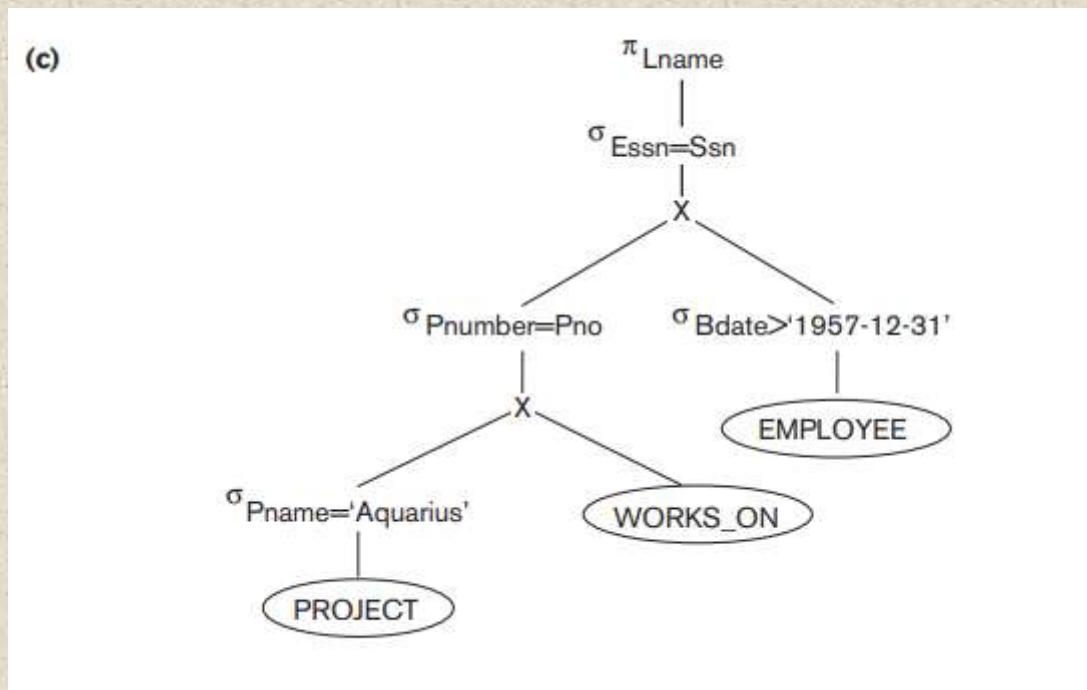
# Outline of a Heuristic Algebraic Optimization Algorithm

- ♦ In the example, Figure 19.2(b) shows the tree in Figure 19.2(a) after applying steps 1 and 2 of the algorithm;



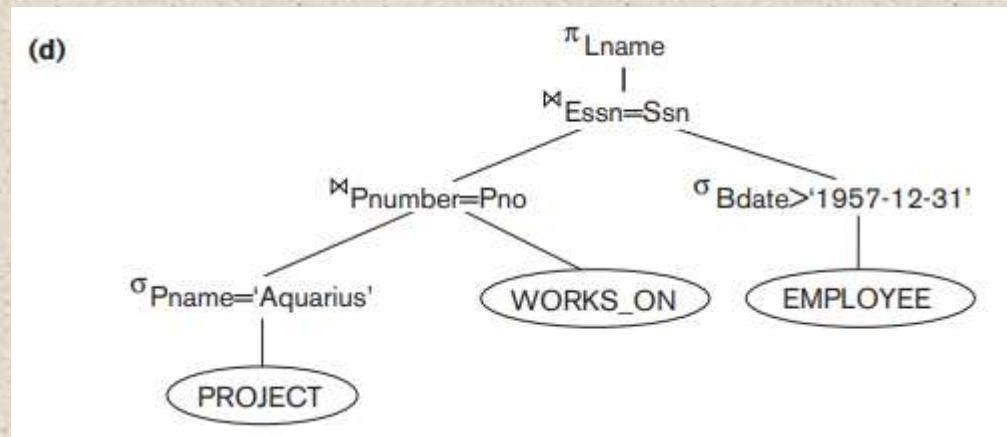
# Outline of a Heuristic Algebraic Optimization Algorithm

- ◆ Figure 19.2(c) shows the tree after step 3;



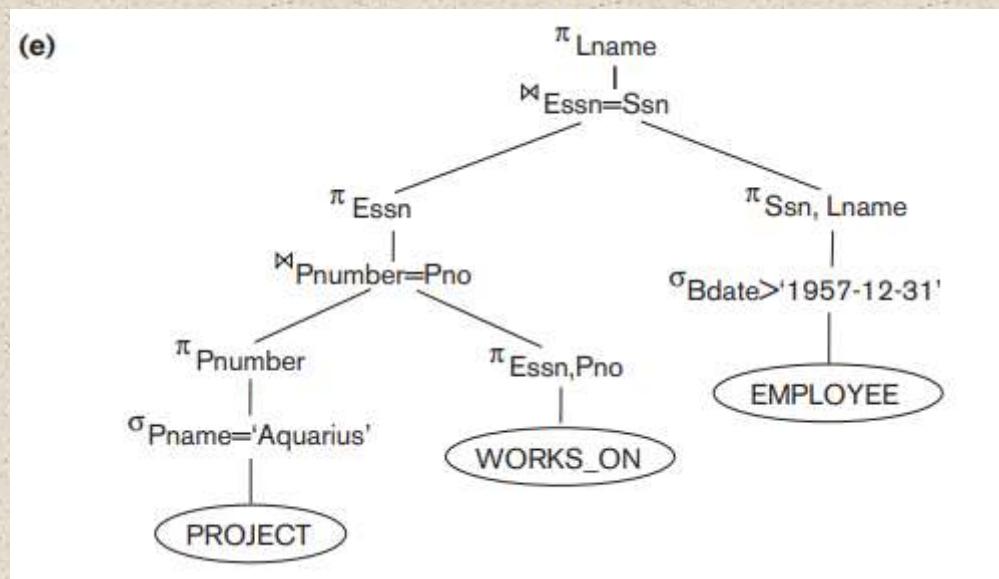
# Outline of a Heuristic Algebraic Optimization Algorithm

◆ Figure 19.2(d) after step 4;



# Outline of a Heuristic Algebraic Optimization Algorithm

◆ Figure 19.2(e) after step 5.



# Outline of a Heuristic Algebraic Optimization Algorithm

- ♦ In step 6, we may group together the operations in the subtree whose root is the operation  $\pi_{E_{SSN}}$  into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation  $\pi_{E_{SSN}}$ , because the first grouping means that this subtree is executed first.

# Outline of a Heuristic Algebraic Optimization Algorithm

- ◆ The main heuristic is to apply first the operations that reduce the size of intermediate results.
- ◆ This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible.
- ◆ Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately

# Cost-Based Optimization

- ◆ A query optimizer does not depend solely on heuristic rules or query transformations; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the lowest cost estimate. . For this approach to work, accurate cost estimates are required so that different strategies can be compared fairly and realistically.
- ◆ It uses traditional optimization techniques that search the solution space to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one.

# **Cost-Based Optimization**

- ♦ Cost-based query optimization is an overall process of choosing the most efficient means of executing a SQL statement based on overall cost of the query. The efficient execution is the execution with minimum cost. To find the cost of query execution plan, the optimization technique uses database statistics.

# Cost Components for Query Execution

◆ **Access cost to secondary storage.** This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as disk I/O (input/output) cost. The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

# Cost Components for Query Execution

- ◆ **Disk storage cost.** This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.
- ◆ **Computation cost.** This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as CPU (central processing unit) cost.

# Cost Components for Query Execution

- ◆ **Memory usage cost.** This is the cost pertaining to the number of main memory buffers needed during query execution.
- ◆ **Communication cost.** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases, it would also include the cost of transferring tables and results among various computers during query evaluation.

# Cost Components for Query Execution

- ◆ For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory buffers.
- ◆ For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost.
- ◆ In distributed databases, where many sites are involved, communication cost must be minimized. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components.
- ◆ This is why some cost functions consider a single factor only—disk access.

# Cost Based Optimization

- ◆ Consider two relations
- ◆ employee (emp\_no, emp\_name, emp\_address, position, salary, branch\_no)
- ◆ (branch\_no, branch\_city, branch\_address, city) with a member of employee can only work at one branch.
- ◆ Consider an SQL query as given below.

SELECT \* FROM Employee, Branch

WHERE Employee.branch\_no = Branch.branch\_no AND  
Employee.position = 'Manager' AND Branch.city = 'Kathmandu';

# Cost Based Optimization

♦SELECT \* FROM Employee, Branch  
WHERE Employee.branch\_no = Branch.branch\_no AND  
Employee.position = 'Manager' AND Branch.city = 'Kathmandu';

**Query 1:**  $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')} (Employee \times Branch)$

**Query 2:**  $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')} (Employee \bowtie Branch)$

**Query 3:**  $(\sigma_{position = 'Manager'} (Employee)) \bowtie (\sigma_{city = 'Kathmandu'} (Branch))$

# Cost Based Optimization

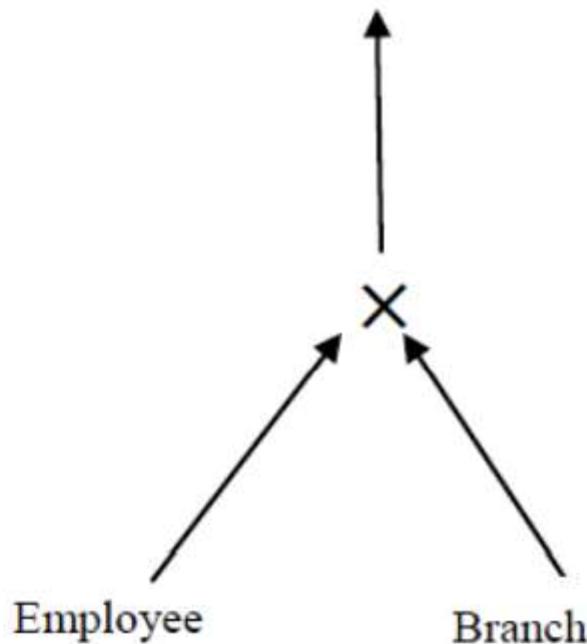
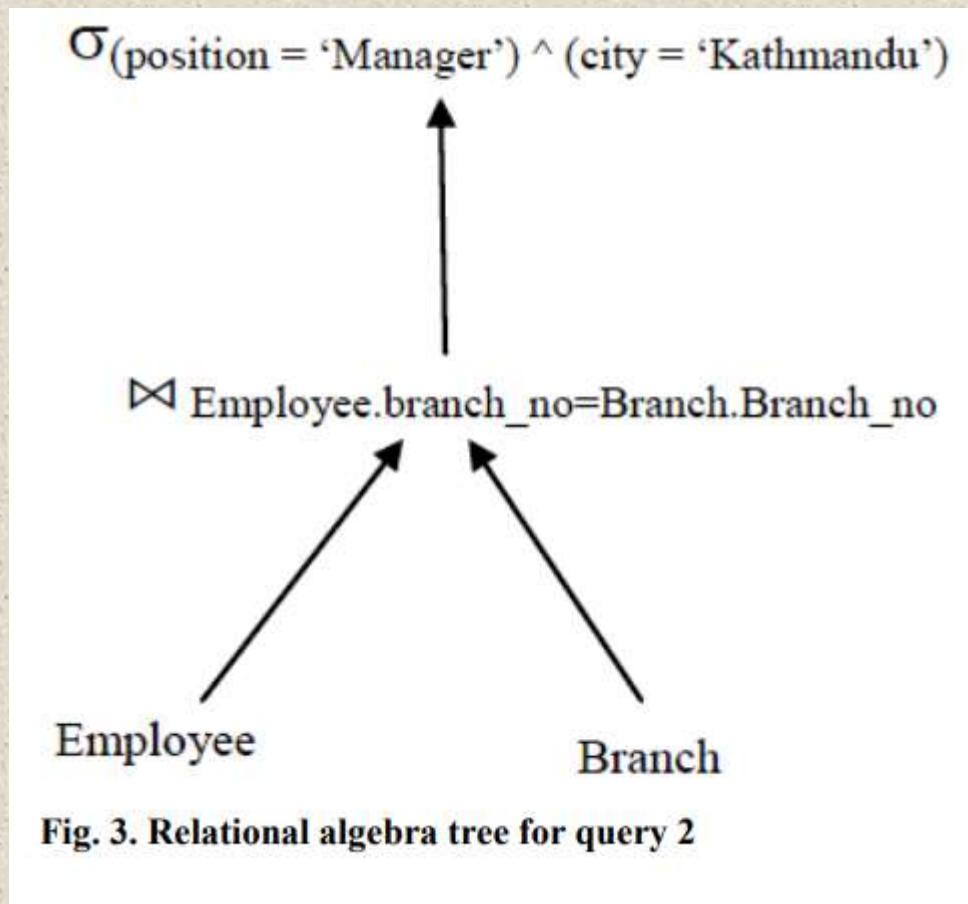
$$\sigma_{(\text{position} = \text{'Manager'}) \wedge (\text{city} = \text{'Kathmandu'}) \wedge (\text{Employee.branch\_no} = \text{Branch.branch\_no})}$$


Fig. 2. Relational algebra tree for query 1

# Cost Based Optimization



# Cost Based Optimization

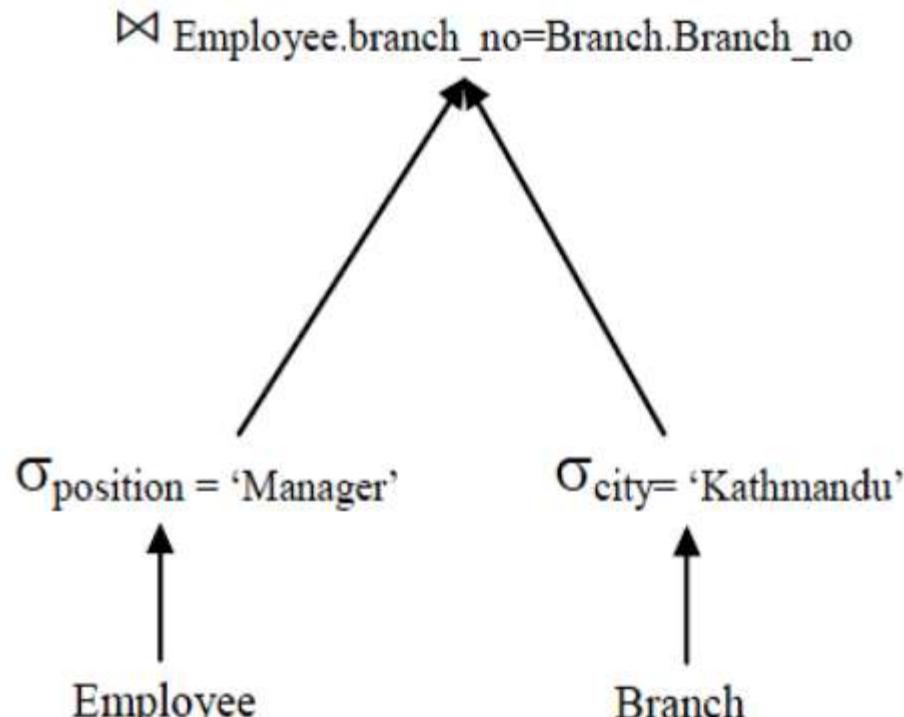


Fig. 4. Relational algebra tree for query 3

# Cost Based Optimization

- ◆ Suppose there are 2000 tuples in Employee, 20 tuples in Branch, 20 Managers (one for each branch), and 10 Kathmandu branches.
- ◆ To compare these three queries, we assume number of disk accesses. We also assume that there are no indexes or sort keys on either relation.
- ◆ The results of any intermediate operations are stored on disk.
- ◆ The cost of the final write is ignored because it is the same in each query.
- ◆ We further assume that tuples are accessed one at a time (although in practice disk accesses would be based on blocks, which would typically contain several tuples), and main memory is large enough to process entire relations for each relational algebra operation.

# Cost Based Optimization

- ◆ The query1 calculates the Cartesian product of Employee and Branch, which requires  $(2000 + 20)$  disk accesses to read these two relations, and creates a relation with  $(2000 \times 20)$  tuples. We then have to read each of these tuples again to test them against the selection predicate at a cost of another  $(2000 \times 20)$  disk accesses, giving a total cost of  $(2000 + 20) + 2 \times (2000 \times 20) = 82020$  disk accesses.

**Query 1:**  $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')} (Employee \bowtie Branch)$

**Query 2:**  $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')} (Employee \bowtie Branch)$

**Query 3:**  $(\sigma_{position = 'Manager'} (Employee)) \bowtie (\sigma_{city = 'Kathmandu'} (Branch))$

# Cost Based Optimization

- ◆ The query 2 joins Employee and Branch which again requires  $(2000 + 20)$  disk accesses to read each of the relations. The Join of these two relations has 2000 tuples, one for each member of Employee. Consequently, the Selection operation requires 2000 disk accesses to read the result of the join, giving a total cost of  $(2000 + 20) + 2 \times (2000) = 6020$  disk accesses.

**Query 1:**  $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')} (Employee \times Branch)$

**Query 2:**  $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')} (Employee \bowtie Branch)$

**Query 3:**  $(\sigma_{position = 'Manager'} (Employee)) \bowtie (\sigma_{city = 'Kathmandu'} (Branch))$

# Cost Based Optimization

- ◆ The query first reads each Employee tuple to determine the Manager tuples, which requires 2000 disk accesses and produces a relation with 20 tuples. Similarly, the second Selection operation reads each Branch tuple to determine the Kathmandu branches, which requires 20 disk accesses and produces a relation with 10 tuples. The final operation is the join of the reduced Employee and Branch relations, which requires  $(20 + 10)$  disk accesses, giving a total cost of  $(2000 + 20) + (20 + 10) + (20 + 10) = 2080$  disk accesses.

# **Cost Based Optimization**

- ◆ From the calculations above, it is clear that query 3 is the most efficient query and is 2.89 times faster than query 2 and 39.43 times faster than the query1.

# **Database Management System (MDS 505)**

## **Jagdish Bhatta**

# Unit – 5.1

# Transaction Processing

# Single User Vs. Multi-User System

- ◆ One criterion for classifying a database system is according to the number of users who can use the system **concurrently**. A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of users and travel agents concurrently.
- ◆ Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system.

# 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—these can include insertion, deletion, modification (update), or retrieval operations.
- ◆ The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries.
- ◆ If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ The *database model* that is used to present transaction processing concepts is simple when compared to the data models that we discussed earlier, such as the relational model or the object model.
- ◆ A **database** is basically represented as a collection of *named data items*. The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general.

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. If the item granularity is a single record, then the record id can be the item name.

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ The basic database access operations that a transaction can include are as follows:
  - **read\_item( $X$ )**. Reads a database item named  $X$  into a program variable. To simplify our notation, we assume that *the program variable is also named  $X$* .
  - **write\_item( $X$ )**. Writes the value of program variable  $X$  into the database item named  $X$ .

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ 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 memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
  3. Copy item  $X$  from the buffer to the program variable 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 memory (if that disk block is not already in some main memory buffer).
  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 (either immediately or at some later point in time).

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ In `write_item(X)` command, It is step 4 that actually updates the database on disk. Sometimes the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.
- ◆ The DBMS will maintain in the **database cache** a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of 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 to choose which of the current occupied buffers is to be replaced.

# Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- ◆ A transaction includes `read_item` and `write_item` operations to access and update the database. Following shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of  $T_1$  in the figure is  $\{X, Y\}$  and its write-set is also  $\{X, Y\}$ .

| (a) | $T_1$                                                                                                                                                                                                        | (b) | $T_2$                                                                                                |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|------------------------------------------------------------------------------------------------------|
|     | <code>read_item(<math>X</math>);</code><br>$X := X - N;$<br><code>write_item(<math>X</math>);</code><br><code>read_item(<math>Y</math>);</code><br>$Y := Y + N;$<br><code>write_item(<math>Y</math>);</code> |     | <code>read_item(<math>X</math>);</code><br>$X := X + M;$<br><code>write_item(<math>X</math>);</code> |

**Figure 20.2**  
Two sample transactions.  
(a) Transaction  $T_1$ .  
(b) Transaction  $T_2$ .

# Why Concurrency Control Is Needed?

- ◆ **Concurrency control and recovery mechanisms** are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items.
- ◆ If this concurrent execution is *uncontrolled*, it may lead to problems, such as an inconsistent database. In the next section, we informally introduce some of the problems that may occur.

# Why Concurrency Control Is Needed?

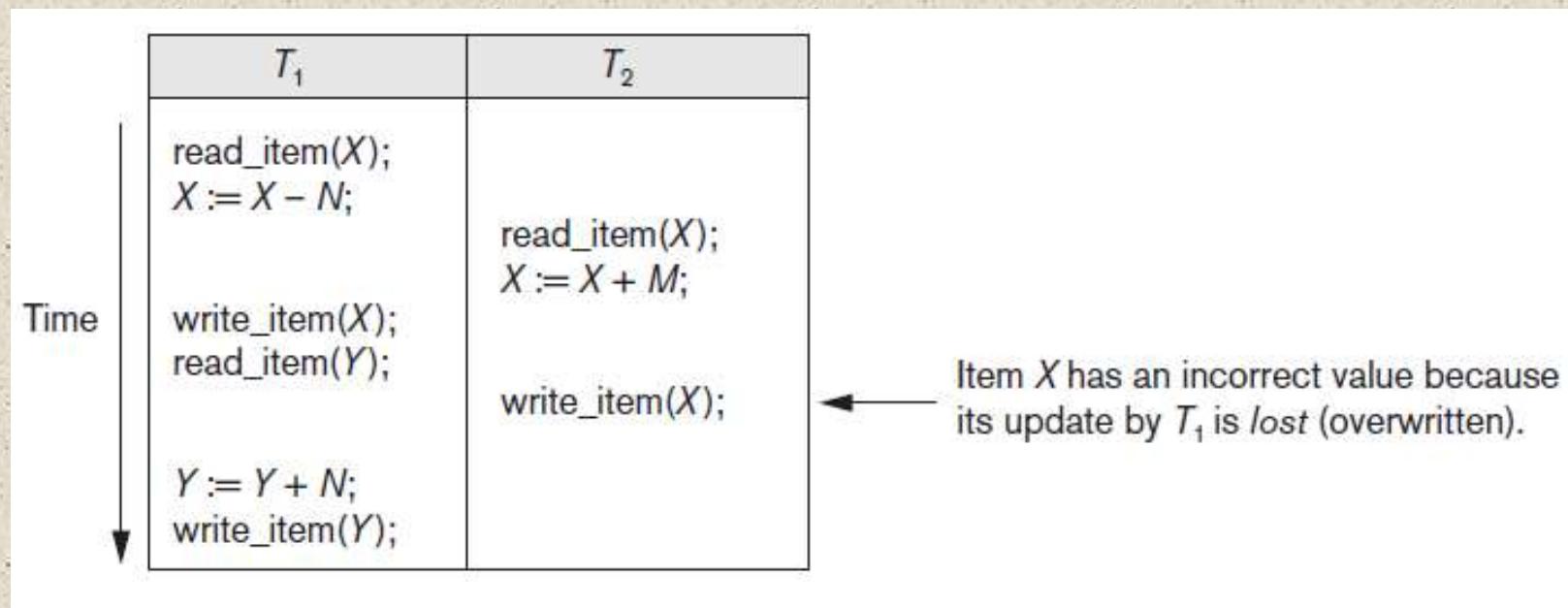
- ◆ Generally, database system allows multiple transactions to run concurrently. Concurrent execution of transaction in database system improves database system performance, reducing transaction waiting time to proceed. It improves resource utilization. But it may lead the database in inconsistent state due to interference among actions of concurrent transactions. Concurrent execution of transaction in database system leads several concurrency control problems.

# Why Concurrency Control is needed?

- ◆ With concurrent execution of transactions, if they are allowed to execute in uncontrolled manner several problems may occur such as;
- ◆ **The Lost Update Problem:** This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

# Why Concurrency Control is needed?

- ◆ **The Lost Update Problem:** Suppose that transactions  $T_1$  and  $T_2$  are submitted at approximately the same time, and suppose that their operations are interleaved as shown in following figure; then the final value of item  $X$  is incorrect because  $T_2$  reads the value of  $X$  *before*  $T_1$  changes it in the database, and hence the updated value resulting from  $T_1$  is lost. For example, if  $X = 80$  at the start,  $N = 5$ , and  $M = 4$ , the final result should be  $X = 79$ . However, in the interleaving of operations shown in the figure, it is  $X = 84$  because the update in  $T_1$  that removed the five seats from  $X$  was *lost*.

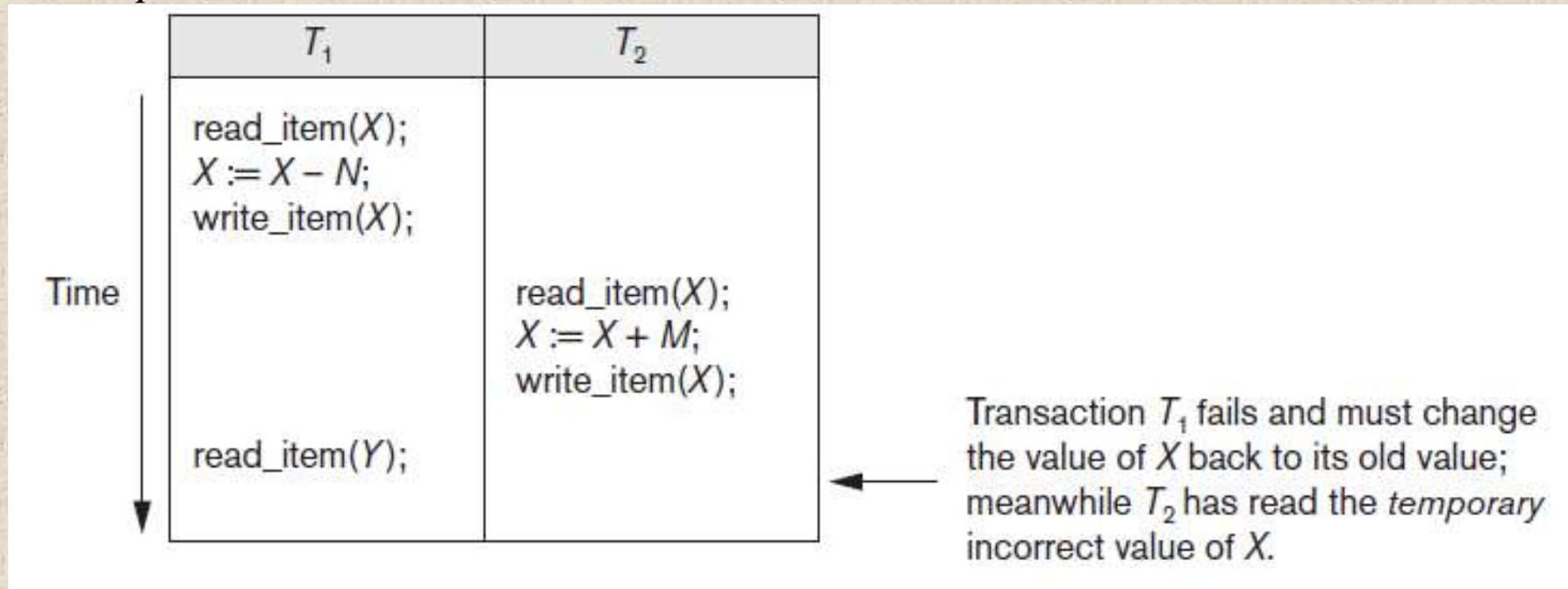


# Why Concurrency Control is needed?

- ◆ **The Temporary Update (or Dirty Read) Problem:** This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

# Why Concurrency Control is needed?

- ◆ **The Temporary Update (or Dirty Read) Problem:** Following Figure shows an example where  $T_1$  updates item  $X$  and then fails before completion, so the system must roll back  $X$  to its original value. Before it can do so, however, transaction  $T_2$  reads the *temporary* value of  $X$ , which will not be recorded permanently in the database because of the failure of  $T_1$ . The value of item  $X$  that is read by  $T_2$  is called *dirty data* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.



# Why Concurrency Control is needed?

- ◆ **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.

# Why Concurrency Control is needed?

- ◆ **The Incorrect Summary Problem:** For example, suppose that a transaction  $T_3$  is calculating the sum; meanwhile, transaction  $T_1$  is executing. If the interleaving of operations shown in following figure occurs, the result of  $T_3$  will be off by an amount  $N$  because  $T_3$  reads the value of  $X$  *after*  $N$  have been subtracted from it but reads the value of  $Y$  *before* those  $N$  have been added to it.

| $T_1$                                               | $T_3$                                                                                                                                                                                                                                                                    |
|-----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>read_item(X); X := X - N; write_item(X);</pre> | <pre>sum := 0; read_item(A); sum := sum + A; ⋮</pre>                                                                                                                                                                                                                     |
| <pre>read_item(Y); Y := Y + N; write_item(Y);</pre> | <pre>read_item(X); sum := sum + X; read_item(Y); sum := sum + Y;</pre> <p>← <math>T_3</math> reads <math>X</math> after <math>N</math> is subtracted and reads <math>Y</math> before <math>N</math> is added; a wrong summary is the result (off by <math>N</math>).</p> |

# Why Concurrency Control is needed?

- ◆ The Unrepeatable read problem:
- ◆ Another problem that may occur is called *unrepeatable read*, where a transaction  $T$  reads the same item twice and the item is changed by another transaction  $T'$  between the two reads. Hence,  $T$  receives *different values* for its two reads of the same item.
- ◆ This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

# Why Recovery Is Needed?

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**. The DBMS must not permit some operations of a transaction  $T$  to be applied to the database while other operations of  $T$  are not, because *the whole transaction* is a logical unit of database processing. If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

# Why Recovery is needed?

- ◆ **Types of Failures.** Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:
  1. **A computer failure (system crash):** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
  2. **A transaction or system error:** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution.
  3. **Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

# Why Recovery is needed?

## ◆ Types of Failures:

4. **Concurrency control enforcement.** The concurrency control method may abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.
  5. **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
  6. **Physical problems and catastrophes.** 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.
- ◆ Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

# Transaction and System Concepts

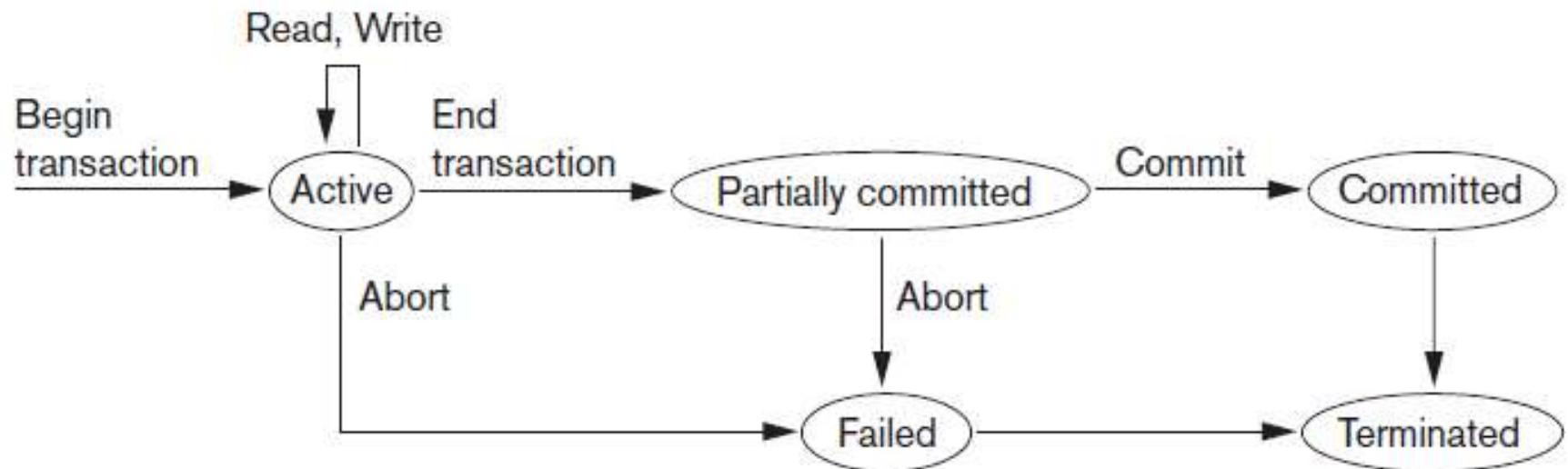
## ◆ Transaction States and Operations

◆ A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts. Therefore, 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 that are executed as part of a transaction.
- END\_TRANSACTION. This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.
- COMMIT\_TRANSACTION. This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- ROLLBACK (or ABORT). This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be **undone**.

# Transaction and System Concepts

## ◆ Transaction States and Operations



**Figure 20.4**

State transition diagram illustrating the states for transaction execution.

# Transaction and System Concepts

- ◆ **Transaction States and Operations**
- ◆ A state transition diagram that illustrates how a transaction moves through its execution states. 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 types of concurrency control protocols may do additional checks to see if the transaction can be committed or not. Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log).
- ◆ If these checks are successful, the transaction is said to have reached its commit point and enters the **committed state**. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

# Transaction and System Concepts

- ◆ **Transaction States and Operations**
- ◆ However, 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 may then have to be rolled back to undo the effect of its WRITE operations on the database.
- ◆ The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions

# Transaction and System Concepts

- ◆ **The System Log**
- ◆ To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the log main memory buffer. When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures.

# Transaction and System Concepts

- ◆ **The System Log**
- ◆ The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record. In these entries,  $T$  refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:
  1. **[start\_transaction,  $T$ ]**. Indicates that transaction  $T$  has started execution.
  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 affirms that its effect can be committed (recorded permanently) to the database.
  5. **[abort,  $T$ ]**. Indicates that transaction  $T$  has been aborted.

# Transaction and System Concepts

- ◆ **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. The transaction then writes a commit record [commit,  $T$ ] into the log.
- ◆ If a system failure occurs, we can search back in the log for all transactions  $T$  that have written a [start\_transaction,  $T$ ] record into the log but have not written their [commit,  $T$ ] record yet; these transactions may have to be *rolled back* to *undo their effect* on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

# Desirable Properties of Transactions: ACID properties

- ◆ **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- ◆ The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

# Desirable Properties of Transactions: ACID properties

- ◆ **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.
- ◆ The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

# Desirable Properties of Transactions: ACID properties

- ◆ **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
- ◆ The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems.
- ◆ **Levels of Isolation.** There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads.

# Desirable Properties of Transactions: ACID properties

- ◆ **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure. The *durability property* is the responsibility of the *recovery subsystem* of the DBMS.

# Characterizing Schedules Based on Recoverability

- ◆ **Schedules (Histories) of Transactions:** When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from all the various transactions is known as a transaction schedule (or history).
- ◆ A **schedule** (or **history**)  $S$  of  $n$  transactions  $T_1, T_2, \dots, T_n$  is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule  $S$ . However, 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  $T_i$ . The order of operations in  $S$  is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.

# Characterizing Schedules Based on Recoverability

- ◆ **Schedules (Histories) of Transactions:**
- ◆ 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, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule. In this notation, the database item  $X$  that is read or written follows the  $r$  and  $w$  operations in parentheses.

# Characterizing Schedules Based on Recoverability

## ◆ Schedules (Histories) of Transactions:

(a)

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

Time ↓

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

(b)

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

Time ↓

Sb: r1(X); w1(X); r2(X); w2(X); r1(Y); a1; [Here it is assumed T1 is aborted after read\_item(Y)]

# Characterizing Schedules Based on Recoverability

- ◆ **Schedules (Histories) of Transactions:**
- ◆ **Conflicting Operations in a Schedule.** Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
  - they belong to *different transactions*;
  - they access the *same item X*; and
  - *at least one* of the operations is a `write_item(X)`.
- ◆ For example, in schedule *S<sub>a</sub>*, the operations  $r1(X)$  and  $w2(X)$  conflict, as do the operations  $r2(X)$  and  $w1(X)$ , and the operations  $w1(X)$  and  $w2(X)$ . However, the operations  $r1(X)$  and  $r2(X)$  do not conflict, since they are both read operations; the operations  $w2(X)$  and  $w1(Y)$  do not conflict because they operate on distinct data items  $X$  and  $Y$ ; and the operations  $r1(X)$  and  $w1(X)$  do not conflict because they belong to the same transaction.

# Characterizing Schedules Based on Recoverability

- ◆ **Schedules (Histories) of Transactions:**
- ◆ Intuitively, two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations  $r1(X); w2(X)$  to  $w2(X); r1(X)$ , then the value of  $X$  that is read by transaction  $T1$  changes, because in the second ordering the value of  $X$  is read by  $r1(X)$  *after* it is changed by  $w2(X)$ , whereas in the first ordering the value is read *before* it is changed. This is called a **read-write conflict**.
- ◆ The other type is called a **write-write conflict** and is illustrated by the case where we change the order of two operations such as  $w1(X); w2(X)$  to  $w2(X); w1(X)$ . For a write-write conflict, the *last value* of  $X$  will differ because in one case it is written by  $T2$  and in the other case by  $T1$ . Notice that two read operations are not conflicting because changing their order makes no difference in outcome.

# Characterizing Schedules Based on Recoverability

- ◆ **Schedules (Histories) of Transactions:**
- ◆ A schedule  $S$  of  $n$  transactions  $T_1, T_2, \dots, T_n$  is said to be a **complete schedule** if the following conditions hold:
  1. The operations in  $S$  are exactly those operations in  $T_1, T_2, \dots, T_n$ , 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  $T_i$ , their relative order of appearance in  $S$  is the same as their order of appearance in  $T_i$ .
  3. For any two conflicting operations, one of the two must occur before the other in the schedule.

# Characterizing Schedules Based on Recoverability

- ◆ **Schedules (Histories) of Transactions:**
- ◆ In general, it is difficult to encounter complete schedules in a transaction processing system because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the **committed projection**  $C(S)$  of a schedule  $S$ , which includes only the operations in  $S$  that belong to committed transactions—that is, transactions  $T_i$  whose commit operation  $c_i$  is in  $S$ .

# 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 quite involved.
- ◆ In some cases, it is even not possible to recover correctly after a failure. Hence, it is important to characterize the types of schedules for which *recovery is possible*, as well as those for which *recovery is relatively simple*. These characterizations do not actually provide the recovery algorithm; they only attempt to theoretically characterize the different types of schedules.

# Characterizing Schedules Based on Recoverability

- ◆ First, we would like to ensure that, once a transaction  $T$  is committed, it should *never* be necessary to roll back  $T$ . This ensures that the durability property of transactions is not violated. The schedules that theoretically meet this criterion are called ***recoverable schedules***. A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS. The condition for a **recoverable schedule** is that 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.
- ◆ A transaction  $T$  **reads** from transaction  $T'$  in a schedule  $S$  if some item  $X$  is first written by  $T'$  and later read by  $T$ . In addition,  $T'$  should not have been aborted before  $T$  reads item  $X$ , and there should be no transactions that write  $X$  after  $T'$  writes it and before  $T$  reads it (unless those transactions, if any, have aborted before  $T$  reads  $X$ ).

# Characterizing Schedules Based on Recoverability

- ◆ Consider the schedule  $Sa'$  given below:

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

- ◆  $Sa'$  is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory.

- ◆ However, consider the two (partial) schedules  $Sc$  and  $Sd$  that follow:

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

$Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;$

$Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;$

- ◆  $Sc$  is not recoverable because  $T2$  reads item  $X$  from  $T1$ , but  $T2$  commits before  $T1$  commits. The problem occurs if  $T1$  aborts after the  $c2$  operation in  $Sc$ ; then the value of  $X$  that  $T2$  read is no longer valid and  $T2$  must be aborted *after* it is committed, leading to a schedule that is *not recoverable*. For the schedule to be recoverable, the  $c2$  operation in  $Sc$  must be postponed until after  $T1$  commits, as shown in  $Sd$ . If  $T1$  aborts instead of committing, then  $T2$  should also abort as shown in  $Se$ , because the value of  $X$  it read is no longer valid. In  $Se$ , aborting  $T2$  is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule  $Sc$ .

# Characterizing Schedules Based on Recoverability

- ◆ In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule  $Se$ , where transaction  $T2$  has to be rolled back because it read item  $X$  from  $T1$ , and  $T1$  then aborted.
- ◆ Because cascading rollback can be time-consuming—since numerous transactions can be rolled back—it is important to characterize the schedules where this phenomenon is guaranteed not to occur. 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. In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur. To satisfy this criterion, the  $r2(X)$  command in schedules  $Sd$  and  $Se$  must be postponed until after  $T1$  has committed (or aborted), thus delaying  $T2$  but ensuring no cascading rollback if  $T1$  aborts.

# Characterizing Schedules Based on Recoverability

- ◆ A **strict schedule**, in which transactions can *neither read nor write* an item  $X$  until the last transaction that wrote  $X$  has committed (or aborted). Strict schedules simplify the recovery process. In a strict schedule, the process of undoing a `write_item( $X$ )` operation of an aborted transaction is simply to restore the **before image** (`old_value` or BFIM) of data item  $X$ . This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. For example, consider schedule  $S_f$ :

$S_f: w1(X, 5); w2(X, 8); a1;$

- ◆ Suppose that the value of  $X$  was originally 9, which is the before image stored in the system log along with the  $w1(X, 5)$  operation. If  $T_1$  aborts, as in  $S_f$ , the recovery procedure that restores the before image of an aborted write operation will restore the value of  $X$  to 9, even though it has already been changed to 8 by transaction  $T_2$ , thus leading to potentially incorrect results. Although schedule  $S_f$  is cascadeless, it is not a strict schedule, since it permits  $T_2$  to write item  $X$  even though the transaction  $T_1$  that last wrote  $X$  had not yet committed (or aborted). A strict schedule does not have this problem.

# Characterizing Schedules Based on Recoverability

- ◆ It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable. Suppose we have  $i$  transactions  $T_1, T_2, \dots, T_i$ , and their number of operations are  $n_1, n_2, \dots, n_i$ , respectively. If we make a set of *all possible schedules* of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

# Characterizing Schedules Based on Serializability

- ◆ The schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*.
- ◆ Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions  $T_1$  and  $T_2$ , in figure shown below, at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:
  - Execute all the operations of transaction  $T_1$  (in sequence) followed by all the operations of transaction  $T_2$  (in sequence).
  - Execute all the operations of transaction  $T_2$  (in sequence) followed by all the operations of transaction  $T_1$  (in sequence).

**Figure 20.2**

Two sample transactions.

(a) Transaction  $T_1$ .  
(b) Transaction  $T_2$ .

(a)

| $T_1$                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------|
| read_item( $X$ );<br>$X := X - N$ ;<br>write_item( $X$ );<br>read_item( $Y$ );<br>$Y := Y + N$ ;<br>write_item( $Y$ ); |

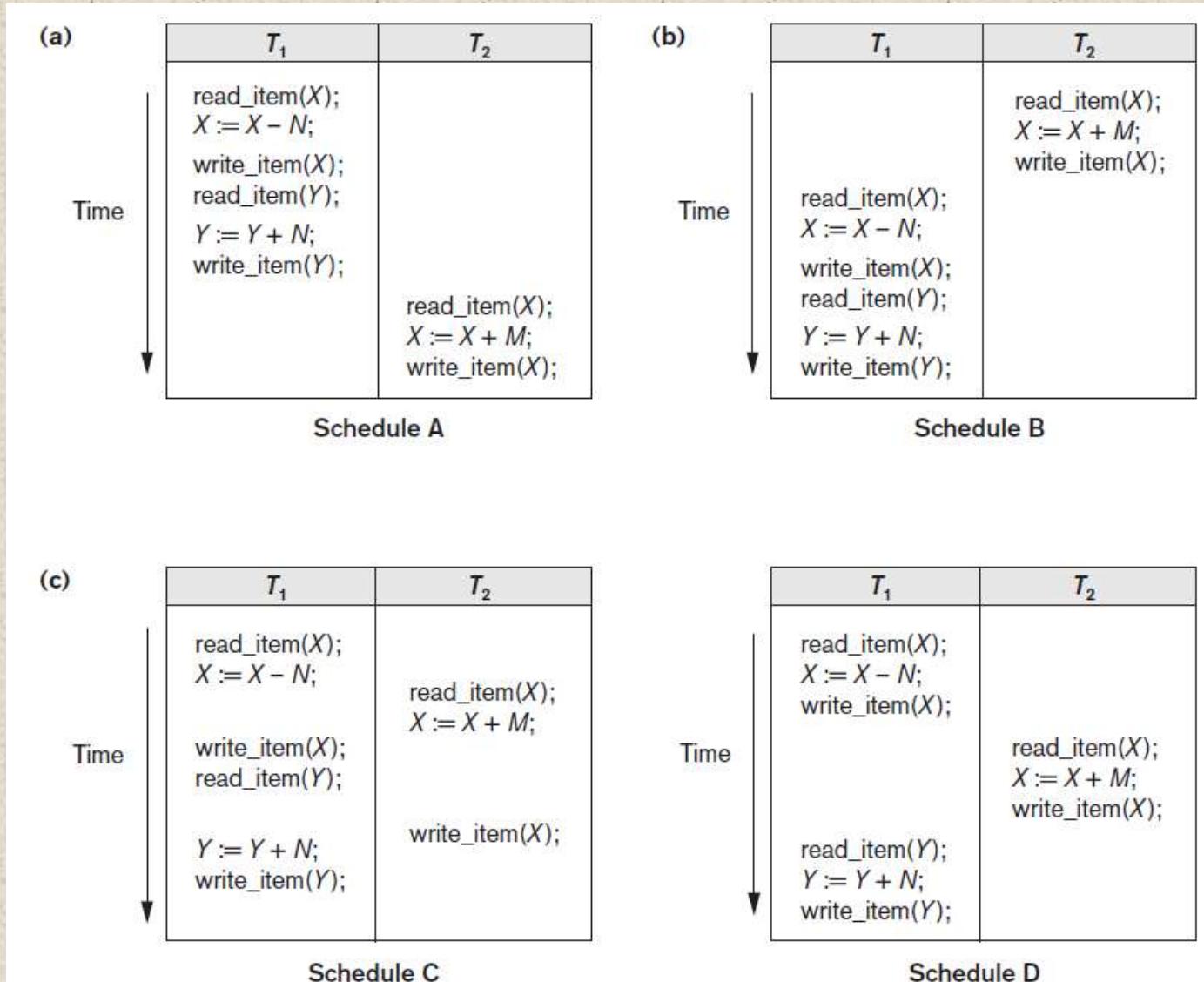
(b)

| $T_2$                                                     |
|-----------------------------------------------------------|
| read_item( $X$ );<br>$X := X + M$ ;<br>write_item( $X$ ); |

# Characterizing Schedules Based on Serializability

- ◆ Considering the transactions from figure 20.2, in previous slide.
- ◆ The two schedules—called *serial schedules*—are shown in figures in next slide Figure 20.5(a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown in Figure 20.5(c). 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.

# Characterizing Schedules Based on Serializability



**Figure 20.5**

Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ . (a) Serial schedule A:  $T_1$  followed by  $T_2$ . (b) Serial schedule B:  $T_2$  followed by  $T_1$ . (c) Two nonserial schedules C and D with interleaving of operations.

# Characterizing Schedules Based on Serializability

- ◆ **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—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*. We can assume this because every transaction is assumed to be correct if executed on its own (according to the *consistency preservation* property). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result.
- ◆ Schedules A and B in the figure in previous slide, Figures 20.5(a) and (b), are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order:  $T_1$  and then  $T_2$  in the figure 20.5(a), and  $T_2$  and then  $T_1$  in Figure 20.5(b). Schedules C and D in figure 20.5(c) are called *nonserial* because each sequence interleaves operations from the two transactions.

# Characterizing Schedules Based on Serializability

- ◆ **Serial, Nonserial, and Conflict-Serializable Schedules**
- ◆ The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, 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. Additionally, if some transaction  $T$  is long, the other transactions must wait for  $T$  to complete all its operations before starting. Hence, serial schedules are *unacceptable* in practice. However, if we can determine which other schedules are *equivalent* to a serial schedule, we can allow these schedules to occur.

# Characterizing Schedules Based on Serializability

- ◆ **Serial, Nonserial, and Conflict-Serializable Schedules**
- ◆ A schedule  $S$  of  $n$  transactions is **serializable** if it is *equivalent to some serial schedule* of the same  $n$  transactions. We will define the concept of *equivalence of schedules* shortly. Notice that there are  $n!$  possible serial schedules of  $n$  transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules— those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to *any* serial schedule and hence are not serializable.
- ◆ Saying that a nonserial schedule  $S$  is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct.

# Characterizing Schedules Based on Serializability

- ◆ **Serial, Nonserial, and Conflict-Serializable Schedules**
- ◆ The simplest but least satisfactory definition involves comparing the effects of the schedules on the database. Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state.
- ◆ Hence, result equivalence alone cannot be used to define equivalence of schedules. The safest and most general approach to defining schedule equivalence is to focus only on the `read_item` and `write_item` operations of the transactions, and not make any assumptions about the other internal operations included in the transactions. For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules *in the same order*.

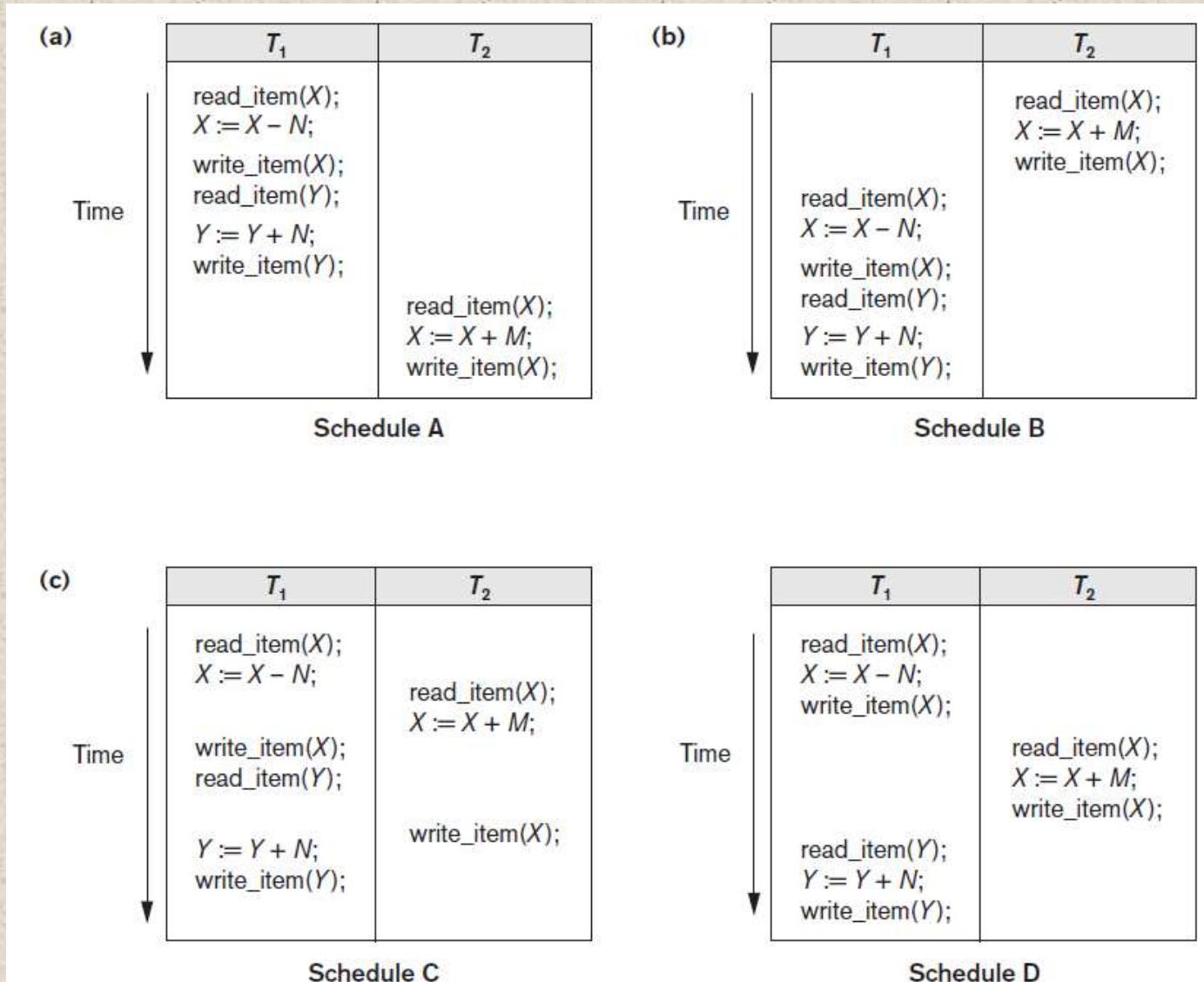
# Characterizing Schedules Based on Serializability

- ◆ **Serial, Nonserial, and Conflict-Serializable Schedules**
- ◆ **Conflict Equivalence of Two Schedules.** Two schedules are said to be **conflict equivalent** if the relative order of any two *conflicting operations* is the same in both schedules.
- ◆ Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and either both are `write_item` operations or one is a `write_item` and the other a `read_item`. If two conflicting operations are applied in *different orders* in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent.
- ◆ For example, if a read and write operation occur in the order  $r1(X)$ ,  $w2(X)$  in schedule  $S1$ , and in the reverse order  $w2(X)$ ,  $r1(X)$  in schedule  $S2$ , the value read by  $r1(X)$  can be different in the two schedules. Similarly, if two write operations occur in the order  $w1(X)$ ,  $w2(X)$  in  $S1$ , and in the reverse order  $w2(X)$ ,  $w1(X)$  in  $S2$ , the next  $r(X)$  operation in the two schedules will read potentially different values; or if these are the last operations writing item  $X$  in the schedules, the final value of item  $X$  in the database will be different.

# Characterizing Schedules Based on Serializability

- ◆ **Serial, Nonserial, and Conflict-Serializable Schedules**
- ◆ **Serializable Schedules.** Using the notion of conflict equivalence, we define a schedule  $S$  to be **serializable** if it is (conflict) 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  $S'$ .

# Characterizing Schedules Based on Serializability



**Figure 20.5**

Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ . (a) Serial schedule A:  $T_1$  followed by  $T_2$ . (b) Serial schedule B:  $T_2$  followed by  $T_1$ . (c) Two nonserial schedules C and D with interleaving of operations.

# Characterizing Schedules Based on Serializability

- ◆ **Serial, Nonserial, and Conflict-Serializable Schedules**
- ◆ Schedule D in Figure 20.5(c), in previous slide, is equivalent to the serial schedule A in Figure 20.5(a). In both schedules, the  $\text{read\_item}(X)$  of  $T_2$  reads the value of  $X$  written by  $T_1$ , whereas the other  $\text{read\_item}$  operations read the database values from the initial database state. Additionally,  $T_1$  is the last transaction to write  $Y$ , and  $T_2$  is the last transaction to write  $X$  in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. Notice that the operations  $r_1(Y)$  and  $w_1(Y)$  of schedule D do not conflict with the operations  $r_2(X)$  and  $w_2(X)$ , since they access different data items. Therefore, we can move  $r_1(Y)$ ,  $w_1(Y)$  before  $r_2(X)$ ,  $w_2(X)$ , leading to the equivalent serial schedule  $T_1, T_2$ .
- ◆ Schedule C in Figure 20.5(c) is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because  $r_2(X)$  and  $w_1(X)$  conflict, which means that we cannot move  $r_2(X)$  down to get the equivalent serial schedule  $T_1, T_2$ . Similarly, because  $w_1(X)$  and  $w_2(X)$  conflict, we cannot move  $w_1(X)$  down to get the equivalent serial schedule  $T_2, T_1$ .

# Characterizing Schedules Based on Serializability

- ◆ **Testing for Serializability of a Schedule**
- ◆ The algorithm for testing serializability looks at only the `read_item` and `write_item` operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph**  $G = (N, E)$  that consists of a set of nodes  $N = \{T_1, T_2, \dots, T_n\}$  and a set of directed edges  $E = \{e_1, e_2, \dots, e_m\}$ .
- ◆ There is one node in the graph for each transaction  $T_i$  in the schedule. Each edge  $e_i$  in the graph is of the form  $(T_j \rightarrow T_k)$ ,  $1 \leq j \leq n$ ,  $1 \leq k \leq n$ , where  $T_j$  is the **starting node** of  $e_i$  and  $T_k$  is the **ending node** of  $e_i$ . Such an edge from node  $T_j$  to node  $T_k$  is created by the algorithm if a pair of conflicting operations exist in  $T_j$  and  $T_k$  and the conflicting operation in  $T_j$  appears in the schedule *before* the *conflicting operation* in  $T_k$ . (Refer slide no [42](#) for conflicting operations)

# Characterizing Schedules Based on Serializability

## ◆ Testing for Serializability of a Schedule

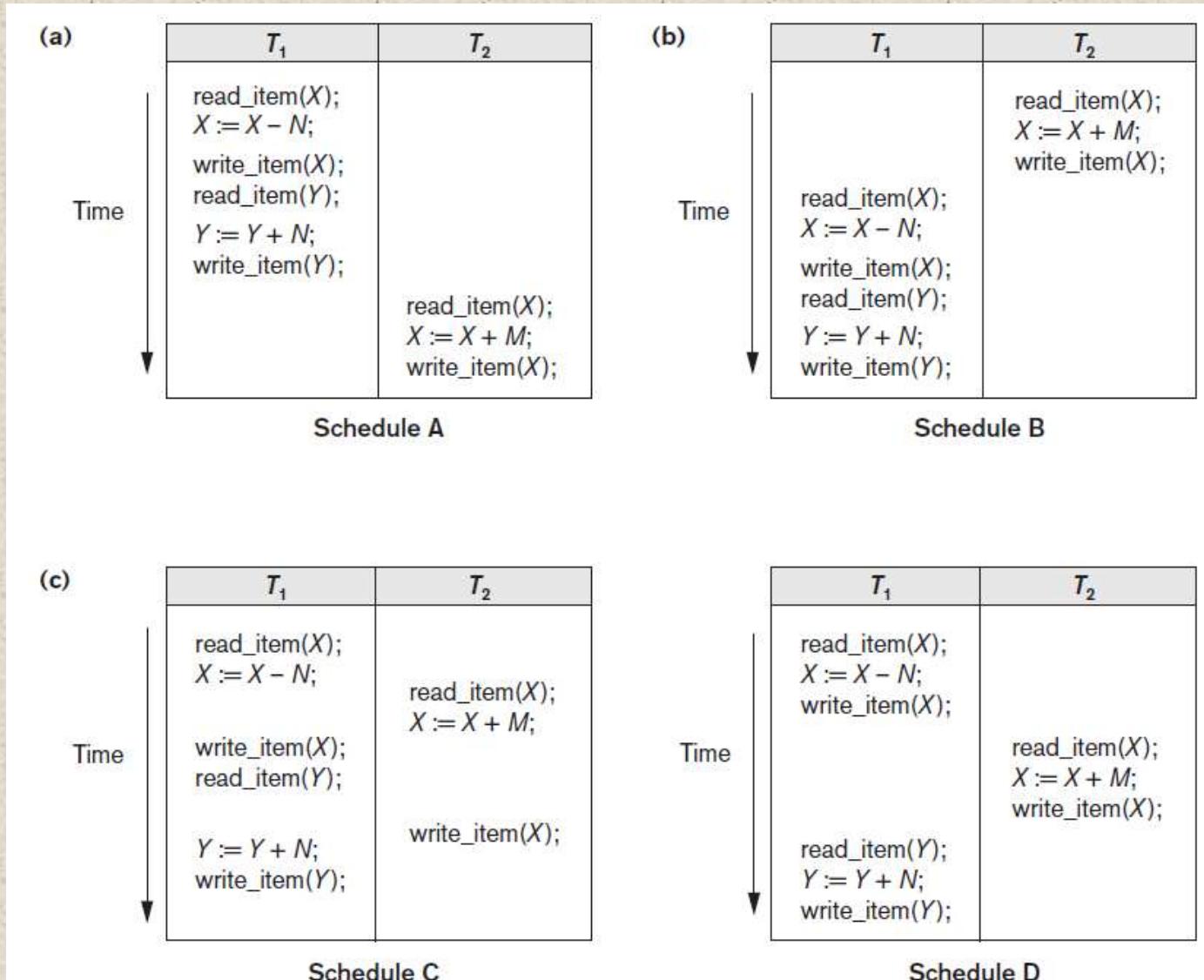
### Algorithm: Testing Conflict Serializability of a Schedule $S$

1. For each transaction  $T_i$  participating in schedule  $S$ , create a node labeled  $T_i$  in the precedence graph.
  2. For each case in  $S$  where  $T_j$  executes a `read_item( $X$ )` after  $T_i$  executes a `write_item( $X$ )`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
  3. For each case in  $S$  where  $T_j$  executes a `write_item( $X$ )` after  $T_i$  executes a `read_item( $X$ )`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
  4. For each case in  $S$  where  $T_j$  executes a `write_item( $X$ )` after  $T_i$  executes a `write_item( $X$ )`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
  5. The schedule  $S$  is serializable if and only if the precedence graph has no cycles
- ◆ If there is a cycle in the precedence graph, schedule  $S$  is not (conflict) serializable; if there is no cycle,  $S$  is (conflict) serializable. A **cycle** in a directed graph is a **sequence of edges**  $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \dots, (T_i \rightarrow T_j))$  with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

# Characterizing Schedules Based on Serializability

- ◆ In the precedence graph, an edge from  $Ti$  to  $Tj$  means that transaction  $Ti$  must come before transaction  $Tj$  in any serial schedule that is equivalent to  $S$ , because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an **equivalent serial schedule  $S'$**  that is equivalent to  $S$ , by ordering the transactions that participate in  $S$  as follows: Whenever an edge exists in the precedence graph from  $Ti$  to  $Tj$ ,  $Ti$  must appear before  $Tj$  in the equivalent serial schedule  $S'$ .
- ◆ 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

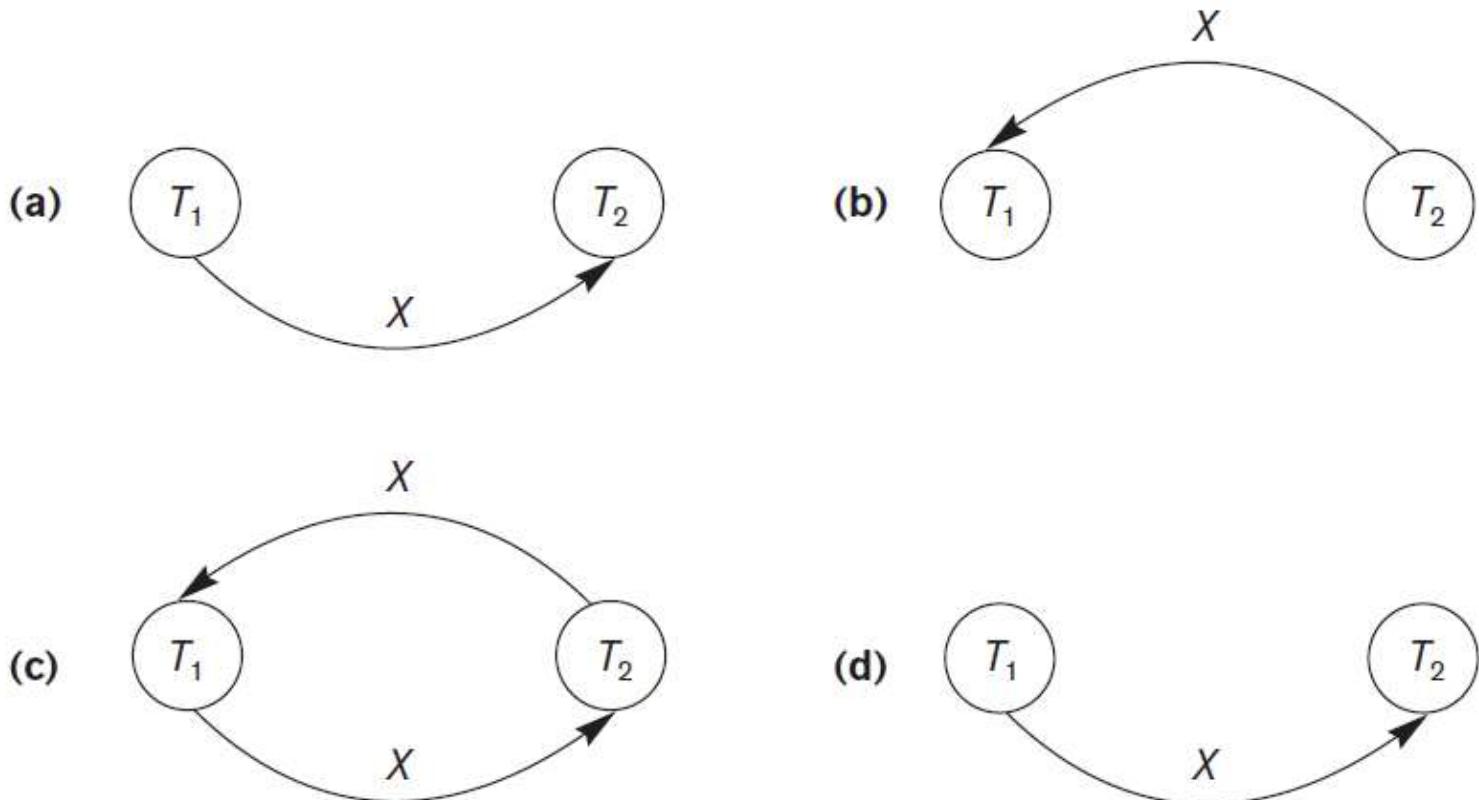
# Characterizing Schedules Based on Serializability



**Figure 20.5**

Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ . (a) Serial schedule A:  $T_1$  followed by  $T_2$ . (b) Serial schedule B:  $T_2$  followed by  $T_1$ . (c) Two nonserial schedules C and D with interleaving of operations.

# Characterizing Schedules Based on Serializability



**Figure 20.7**

Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability. (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).

# Characterizing Schedules Based on Serializability

- ◆ The graph for schedule C has a cycle, so it is not serializable.
- ◆ The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is  $T_1$  followed by  $T_2$ .
- ◆ The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

# How Serializability Is Used for Concurrency Control

- ◆ Saying that a schedule  $S$  is (conflict) serializable—that is,  $S$  is (conflict) equivalent to a serial schedule—is tantamount to saying that  $S$  is correct. Being *serializable* is distinct from being *serial*, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for a long transaction to delay other transactions, thus slowing down transaction processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness.
- ◆ In practice, it is difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates resources to all processes.

# How Serializability Is Used for Concurrency Control

- ◆ Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule.
- ◆ Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability. If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. The approach taken in most commercial DBMSs is to design **protocols** (sets of rules) that—if followed by *every* individual transaction or if enforced by a DBMS concurrency control subsystem—will ensure serializability of *all schedules in which the transactions participate*.

# How Serializability Is Used for Concurrency Control

- ◆ Another problem is that transactions are submitted continuously to the system, so it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule  $S$ . The *committed projection*  $C(S)$  of a schedule  $S$  includes only the operations in  $S$  that belong to committed transactions. We can theoretically define a schedule  $S$  to be serializable if its committed projection  $C(S)$  is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

# **Database Management System (MDS 505)**

## **Jagdish Bhatta**

# Unit – 5.2

# Concurrency Control Techniques

# **Concurrency Control**

- ◆ Generally, database system allows multiple transactions to run concurrently.
- ◆ Concurrent execution of transaction in database system improves database system performance, reducing transaction waiting time to proceed. It improves resource utilization. But it may lead the database in inconsistent state due to interference among actions of concurrent transactions.
- ◆ Concurrent execution of transaction in database system leads several concurrency control problems.

# Purpose of Concurrency Control

- ◆ To enforce isolation (through mutual exclusion) among conflicting transactions.
- ◆ To preserve database consistency through consistency preserving execution of transactions.
- ◆ To resolve read-write and write-write conflicts

# 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 that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database.
- ◆ Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ Several types of locks are used in concurrency control. To introduce locking concepts gradually, there are binary locks, which are simple but are also *too restrictive for database concurrency control purposes* and so are not used much.
- ◆ Another types of locks are *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in database locking schemes.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ **Binary Locks.** A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item  $X$ . If the value of the lock on  $X$  is 1, item  $X$  *cannot be accessed* by a database operation that requests the item. If the value of the lock on  $X$  is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item  $X$  as **lock( $X$ )**.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ Two operations, `lock_item` and `unlock_item`, are used with binary locking. A transaction requests access to an item  $X$  by first issuing a **lock\_item( $X$ )** operation. If  $\text{LOCK}(X) = 1$ , the transaction is forced to wait. If  $\text{LOCK}(X) = 0$ , it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item  $X$ . When the transaction is through using the item, it issues an **unlock\_item( $X$ )** operation, which sets  $\text{LOCK}(X)$  back to 0 (**unlocks** the item) so that  $X$  may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item.

# Two-Phase Locking Techniques for Concurrency Control

## ◆ Types of Locks and System Lock Tables

```
lock_item(X):
B: if $\text{LOCK}(X) = 0$ (*item is unlocked*)
 then $\text{LOCK}(X) \leftarrow 1$ (*lock the item*)
```

```
else
begin
wait (until $\text{LOCK}(X) = 0$
 and the lock manager wakes up the transaction);
go to B
end;
```

```
unlock_item(X):
 $\text{LOCK}(X) \leftarrow 0$; (* unlock the item *)
if any transactions are waiting
then wakeup one of the waiting transactions;
```

**Figure 21.1**  
Lock and unlock operations  
for binary locks.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ Notice that the lock\_item and unlock\_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 21.1 in previous slide, the wait command within the lock\_item( $X$ ) operation is usually implemented by putting the transaction in a waiting queue for item  $X$  until  $X$  is unlocked and the transaction can be granted access to it. Other transactions that also want to access  $X$  are placed in the same queue. Hence, the wait command is considered to be outside the lock\_item operation.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ It is simple to implement a binary lock; all that is needed is a binary-valued variable, **LOCK**, associated with each data item  $X$  in the database. In its simplest form, each lock can be a record with three fields:  $\langle \text{Data\_item\_name}, \text{LOCK}, \text{Locking\_transaction} \rangle$  plus a queue for transactions that are waiting to access the item. The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ If the simple binary locking scheme described here is used, every transaction must obey the following rules:
  1. A transaction  $T$  must issue the operation  $\text{lock\_item}(X)$  before any  $\text{read\_item}(X)$  or  $\text{write\_item}(X)$  operations are performed in  $T$ .
  2. A transaction  $T$  must issue the operation  $\text{unlock\_item}(X)$  after all  $\text{read\_item}(X)$  and  $\text{write\_item}(X)$  operations are completed in  $T$ .
  3. A transaction  $T$  will not issue a  $\text{lock\_item}(X)$  operation if it already holds the lock on item  $X$ .<sup>1</sup>
  4. A transaction  $T$  will not issue an  $\text{unlock\_item}(X)$  operation unless it already holds the lock on item  $X$ .
- ◆ These rules can be enforced by the lock manager module of the DBMS. Between the  $\text{lock\_item}(X)$  and  $\text{unlock\_item}(X)$  operations in transaction  $T$ ,  $T$  is said to **hold the lock** on item  $X$ . At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ **Shared/Exclusive (or Read/Write) Locks.** The preceding binary locking scheme is too restrictive for database items because at most one transaction can hold a lock on a given item.
- ◆ We should allow several transactions to access the same item  $X$  if they all access  $X$  for *reading purposes only*. This is because read operations on the same item by different transactions are *not conflicting*. However, 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**—there are three locking operations:  $\text{read\_lock}(X)$ ,  $\text{write\_lock}(X)$ , and  $\text{unlock}(X)$ . A lock associated with an item  $X$ ,  $\text{LOCK}(X)$ , now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table, as well as a list of transaction ids that hold a shared lock.
- ◆ Each record in the lock table will have four fields: <Data\_item\_name, LOCK, No\_of\_reads, Locking\_transaction(s)>. The system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either readlocked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If  $\text{LOCK}(X)$  = write-locked, the value of locking\_transaction(s) is a *single transaction* that holds the exclusive (write) lock on X. If  $\text{LOCK}(X)$ =read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ The three operations  $\text{read\_lock}(X)$ ,  $\text{write\_lock}(X)$ , and  $\text{unlock}(X)$  are described in Figure 21.2., in next slide. As before, each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

# Two-Phase Locking Techniques for Concurrency Control

```
read_lock(X):
B: if LOCK(X) = "unlocked"
 then begin LOCK(X) ← "read-locked";
 no_of_reads(X) ← 1
 end
 else if LOCK(X) = "read-locked"
 then no_of_reads(X) ← no_of_reads(X) + 1
 else begin
 wait (until LOCK(X) = "unlocked"
 and the lock manager wakes up the transaction);
 go to B
 end;
write_lock(X):
B: if LOCK(X) = "unlocked"
 then LOCK(X) ← "write-locked"
 else begin
 wait (until LOCK(X) = "unlocked"
 and the lock manager wakes up the transaction);
 go to B
 end;
unlock (X):
 if LOCK(X) = "write-locked"
 then begin LOCK(X) ← "unlocked";
 wakeup one of the waiting transactions, if any
 end
 else if LOCK(X) = "read-locked"
 then begin
 no_of_reads(X) ← no_of_reads(X) -1;
 if no_of_reads(X) = 0
 then begin LOCK(X) = "unlocked";
 wakeup one of the waiting transactions, if any
 end
 end
 end;
```

**Figure 21.2**  
Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ When we use the shared/exclusive locking scheme, the system must enforce the following rules:
  1. A transaction  $T$  must issue the operation  $\text{read\_lock}(X)$  or  $\text{write\_lock}(X)$  before any  $\text{read\_item}(X)$  operation is performed in  $T$ .
  2. A transaction  $T$  must issue the operation  $\text{write\_lock}(X)$  before any  $\text{write\_item}(X)$  operation is performed in  $T$ .
  3. A transaction  $T$  must issue the operation  $\text{unlock}(X)$  after all  $\text{read\_item}(X)$  and  $\text{write\_item}(X)$  operations are completed in  $T$ .
  4. A transaction  $T$  will not issue a  $\text{read\_lock}(X)$  operation if it already holds a read (shared) lock or a write (exclusive) lock on item  $X$ . This rule may be relaxed for downgrading of locks, as we discuss shortly.
  5. A transaction  $T$  will not issue a  $\text{write\_lock}(X)$  operation if it already holds a read (shared) lock or write (exclusive) lock on item  $X$ . This rule may also be relaxed for upgrading of locks, as we discuss shortly.
  6. A transaction  $T$  will not issue an  $\text{unlock}(X)$  operation unless it already holds a read (shared) lock or a write (exclusive) lock on item  $X$ .

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ **Conversion (Upgrading, Downgrading) of Locks.** It is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, 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, it is possible for a transaction  $T$  to issue a `read_lock( $X$ )` and then later to **upgrade** the lock by issuing a `write_lock( $X$ )` operation. If  $T$  is the only transaction holding a read lock on  $X$  at the time it issues the `write_lock( $X$ )` operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction  $T$  to issue a `write_lock( $X$ )` and then later to **downgrade** the lock by issuing a `read_lock( $X$ )` operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the `locking_transaction(s)` field) to store the information on which transactions hold locks on the item.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Types of Locks and System Lock Tables**
- ◆ Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 21.3, in the next slide, shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 21.3(a) the items  $Y$  in  $T1$  and  $X$  in  $T2$  were unlocked too early. This allows a schedule such as the one shown in Figure 21.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction.

# Two-Phase Locking Techniques for Concurrency Control

## ◆ Types of Locks and System Lock Tables

(a)

| $T_1$                                                                                                                                                         | $T_2$                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| read_lock( $Y$ );<br>read_item( $Y$ );<br>unlock( $Y$ );<br>write_lock( $X$ );<br>read_item( $X$ );<br>$X := X + Y$ ;<br>write_item( $X$ );<br>unlock( $X$ ); | read_lock( $X$ );<br>read_item( $X$ );<br>unlock( $X$ );<br>write_lock( $Y$ );<br>read_item( $Y$ );<br>$Y := X + Y$ ;<br>write_item( $Y$ );<br>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  $T_1$ :  $X=70, Y=50$

(c)

Time ↓

| $T_1$                                                                                             | $T_2$                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| read_lock( $Y$ );<br>read_item( $Y$ );<br>unlock( $Y$ );                                          | read_lock( $X$ );<br>read_item( $X$ );<br>unlock( $X$ );<br>write_lock( $Y$ );<br>read_item( $Y$ );<br>$Y := X + Y$ ;<br>write_item( $Y$ );<br>unlock( $Y$ ); |
| write_lock( $X$ );<br>read_item( $X$ );<br>$X := X + Y$ ;<br>write_item( $X$ );<br>unlock( $X$ ); |                                                                                                                                                               |

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

**Figure 21.3**

Transactions that do not obey two-phase locking.  
(a) Two transactions  $T_1$  and  $T_2$ . (b) Results of  
possible serial schedules of  $T_1$  and  $T_2$ . (c) A  
nonserializable schedule  $S$  that uses locks.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Guaranteeing Serializability by Two-Phase Locking**
- ◆ A transaction is said to follow the **two-phase locking protocol, often known as basic 2PL**, if *all* locking operations (read\_lock, write\_lock) precede the *first* unlock operation in the transaction. Such a transaction can be divided into two phases: an **expanding or growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

# Two-Phase Locking Techniques for Concurrency Control

## ◆ Guaranteeing Serializability by Two-Phase Locking

| $T_1'$                                                                                                              | $T_2'$                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <pre>read_lock(Y); read_item(Y); write_lock(X); unlock(Y) read_item(X); X := X + Y; write_item(X); unlock(X);</pre> | <pre>read_lock(X); read_item(X); write_lock(Y); unlock(X) read_item(Y); Y := X + Y; write_item(Y); unlock(Y);</pre> |

**Figure 21.4**

Transactions  $T_1'$  and  $T_2'$ , which are the same as  $T_1$  and  $T_2$  in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Guaranteeing Serializability by Two-Phase Locking**
- ◆ It can be proved that, if *every* transaction in a schedule follows the two-phase locking protocol, the schedule is *guaranteed to be serializable*, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.
- ◆ Two-phase locking may limit the amount of concurrency that can occur in a schedule because a transaction  $T$  may not be able to release an item  $X$  after it is through using it if  $T$  must lock an additional item  $Y$  later; or, conversely,  $T$  must lock the additional item  $Y$  before it needs it so that it can release  $X$ . Hence,  $X$  must remain locked by  $T$  until all items that the transaction needs to read or write have been locked; only then can  $X$  be released by  $T$ . Meanwhile, another transaction seeking to access  $X$  may be forced to wait, even though  $T$  is done with  $X$ ; conversely, if  $Y$  is locked earlier than it is needed, another transaction seeking to access  $Y$  is forced to wait even though  $T$  is not using  $Y$  yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Basic, Conservative, Strict, and Rigorous Two-Phase Locking.**
- ◆ There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*.
- ◆ The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a *deadlock-free protocol*. However, it is difficult to use in practice because of the need to predeclare the read-set and writeset, which is not possible in some situations.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Basic, Conservative, Strict, and Rigorous Two-Phase Locking.**
- ◆ In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules. In this variation, a transaction  $T$  does not release any of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by  $T$  unless  $T$  has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free.
- ◆ A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction  $T$  does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Basic, Conservative, Strict, and Rigorous Two-Phase Locking.**
- ◆ Notice the difference between strict and rigorous 2PL: the former holds write-locks until it commits, whereas the latter holds all locks (read and write). Also, the difference between conservative and rigorous 2PL is that the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by committing or aborting), so the transaction is in its expanding phase until it ends.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Basic, Conservative, Strict, and Rigorous Two-Phase Locking.**
- ◆ Usually the **concurrency control subsystem** itself is responsible for generating the `read_lock` and `write_lock` requests.
- ◆ For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction  $T$  issues a `read_item( $X$ )`, the system calls the `read_lock( $X$ )` operation on behalf of  $T$ . If the state of  $\text{LOCK}(X)$  is `write_locked` by some other transaction  $T'$ , the system places  $T$  in the waiting queue for item  $X$ ; otherwise, it grants the `read_lock( $X$ )` request and permits the `read_item( $X$ )` operation of  $T$  to execute.
- ◆ On the other hand, if transaction  $T$  issues a `write_item( $X$ )`, the system calls the `write_lock( $X$ )` operation on behalf of  $T$ . If the state of  $\text{LOCK}(X)$  is `write_locked` or `read_locked` by some other transaction  $T'$ , the system places  $T$  in the waiting queue for item  $X$ ; if the state of  $\text{LOCK}(X)$  is `read_locked` and  $T$  itself is the only transaction holding the read lock on  $X$ , the system upgrades the lock to `write_locked` and permits the `write_item( $X$ )` operation by  $T$ . Finally, if the state of  $\text{LOCK}(X)$  is `unlocked`, the system grants the `write_lock( $X$ )` request and permits the `write_item( $X$ )` operation to execute. After each action, the system must *update its lock table* appropriately.

# **Two-Phase Locking Techniques for Concurrency Control**

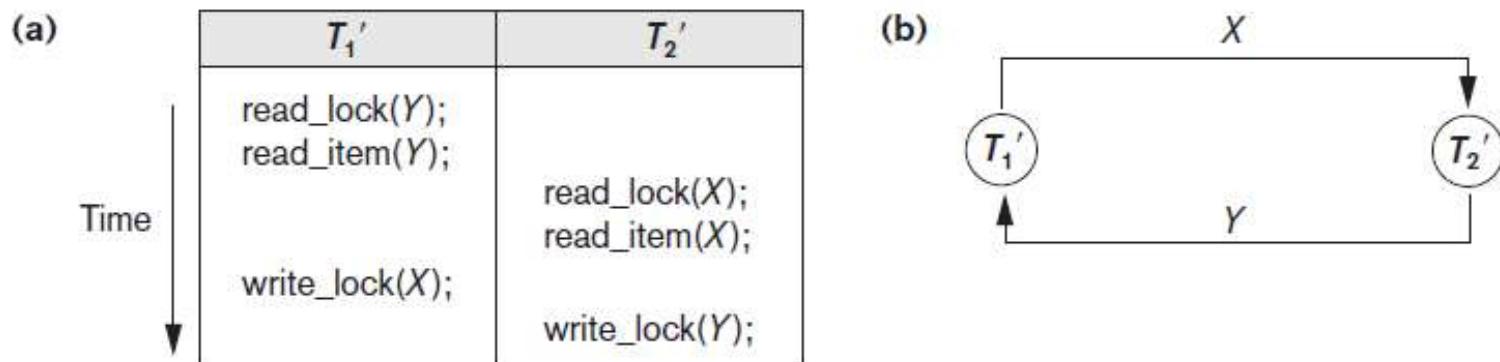
- ◆ **Basic, Conservative, Strict, and Rigorous Two-Phase Locking.**
- ◆ Locking is generally considered to have a high overhead, because every read or write operation is preceded by a system locking request. The use of locks can also cause two additional problems: deadlock and starvation

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Dealing with Deadlock and Starvation**
- ◆ **Deadlock** occurs when *each* transaction  $T$  in a set of *two or more transactions* is waiting for some item that is locked by some other transaction  $T'$  in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock. A simple example is shown in figure 21.5(a), in the next slide, where the two transactions  $T1'$  and  $T2'$  are deadlocked in a partial schedule;  $T1'$  is in the waiting queue for  $X$ , which is locked by  $T2'$ , whereas  $T2'$  is in the waiting queue for  $Y$ , which is locked by  $T1'$ . Meanwhile, neither  $T1'$  nor  $T2'$  nor any other transaction can access items  $X$  and  $Y$ .

# Two-Phase Locking Techniques for Concurrency Control

- ◆ Dealing with Deadlock and Starvation



**Figure 21.5**

Illustrating the deadlock problem. (a) A partial schedule of  $T_1'$  and  $T_2'$  that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Prevention Protocols**
- ◆ One way to prevent deadlock is to use a **deadlock prevention protocol**. One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously, this solution further limits concurrency.
- ◆ A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Prevention Protocols**
- ◆ A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? Some of these techniques use the concept of **transaction timestamp**  $TS(T')$ , which is a unique identifier assigned to each transaction.
- ◆ The timestamps are typically based on the order in which transactions are started; hence, if transaction  $T1$  starts before transaction  $T2$ , then  $TS(T1) < TS(T2)$ . Notice that the *older* transaction (which starts first) has the *smaller* timestamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Prevention Protocols**
- ◆ Suppose that transaction  $Ti$  tries to lock an item  $X$  but is not able to because  $X$  is locked by some other transaction  $Tj$  with a conflicting lock. 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.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Prevention Protocols**
- ◆ In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are *deadlock-free*, since in waitdie, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a deadlock*.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Prevention Protocols**
- ◆ Another group of protocols that prevent deadlock do not require timestamps. These include the **no waiting (NW)** and **cautious waiting (CW) algorithms**.
- ◆ In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Prevention Protocols**
- ◆ The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction  $T_i$  tries to lock an item  $X$  but is not able to do so because  $X$  is locked by some other transaction  $T_j$  with a conflicting lock. The cautious waiting rule is as follows:
  - **Cautious waiting.** If  $T_j$  is not blocked (not waiting for some other locked item), then  $T_i$  is blocked and allowed to wait; otherwise abort  $T_i$ .
- ◆ It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time  $b(T)$  at which each blocked transaction  $T$  was blocked, if the two transactions  $T_i$  and  $T_j$  above both become blocked and  $T_i$  is waiting for  $T_j$ , then  $b(T_i) < b(T_j)$ , since  $T_i$  can only wait for  $T_j$  at a time when  $T_j$  is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

# Two-Phase Locking Techniques for Concurrency Control

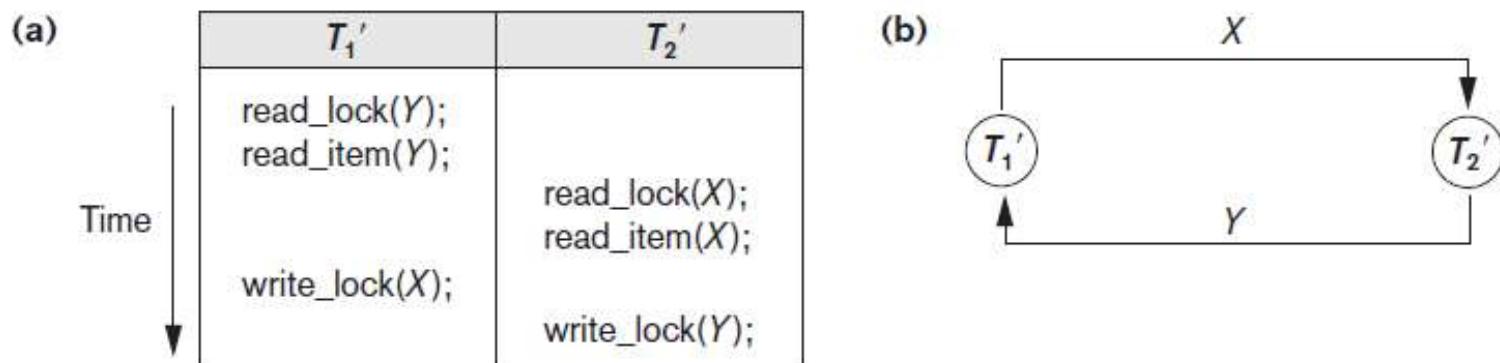
- ◆ **Deadlock Detection**
- ◆ An alternative approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists.
- ◆ This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is heavy, it may be advantageous to use a deadlock prevention scheme.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Detection**
- ◆ A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction  $T_i$  is waiting to lock an item  $X$  that is currently locked by a transaction  $T_j$ , a directed edge  $(T_i \rightarrow T_j)$  is created in the wait-for graph. When  $T_j$  releases the lock(s) on the items that  $T_i$  was waiting for, the directed edge is dropped from the wait-for graph.
- ◆ We have a state of deadlock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. One possibility is to check for a cycle every time an edge is added to the wait for graph, but this may cause excessive overhead. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle.

# Two-Phase Locking Techniques for Concurrency Control

## ◆ Deadlock Detection



**Figure 21.5**

Illustrating the deadlock problem. (a) A partial schedule of  $T_1'$  and  $T_2'$  that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Deadlock Detection**
- ◆ If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. 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 try instead to select transactions that have not made many changes (younger transactions).

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Timeouts**
- ◆ Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, 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 it—regardless of whether a deadlock actually exists.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ Starvation
- ◆ Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.

# Two-Phase Locking Techniques for Concurrency Control

- ◆ **Starvation**
- ◆ Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

# Concurrency Control Based on Timestamp Ordering

- ◆ The use of locking, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem. A different approach to concurrency control involves using **transaction timestamps** to order transaction execution for an equivalent serial schedule.

## Concurrency Control Based on Timestamp Ordering

- ◆ **Timestamps:** 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, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction  $T$  as  $\text{TS}(T)$ .
- ◆ Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.
- ◆ Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

# The Timestamp Ordering Algorithm for Concurrency Control

- ◆ The idea for this scheme is to enforce the equivalent serial order on the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**. Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps.
- ◆ The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of *conflicting operations* in the schedule, the order in which the item is accessed must follow the timestamp order.

# The Timestamp Ordering Algorithm for Concurrency Control

- ◆ The algorithm associates with each database item  $X$  two timestamp (TS) values:
  - **read\_TS( $X$ )**: The **read timestamp** of item  $X$  is the largest timestamp among all the timestamps of transactions that have successfully read item  $X$ —that is,  $\text{read\_TS}(X) = \text{TS}(T)$ , where  $T$  is the *youngest* transaction that has read  $X$  successfully.
  - **write\_TS( $X$ )**: The **write timestamp** of item  $X$  is the largest of all the timestamps of transactions that have successfully written item  $X$ —that is,  $\text{write\_TS}(X) = \text{TS}(T)$ , where  $T$  is the *youngest* transaction that has written  $X$  successfully.

## Multiversion Concurrency Control Techniques

- ◆ These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written); they are known as **multiversion concurrency control** because several versions (values) of an item are kept by the system.
- ◆ When a transaction requests to read an item, the *appropriate* version is chosen to maintain the serializability of the currently executing schedule. One reason for keeping multiple versions is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability.
- ◆ When a transaction writes an item, it writes a *new version* and the old version(s) of the item is retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

# Multiversion Concurrency Control Techniques

- ◆ An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. In some cases, older versions can be kept in a temporary store. It is also possible that older versions may have to be maintained anyway—for example, for recovery purposes.
- ◆ Some database applications may require older versions to be kept to maintain a history of the changes of data item values. The extreme case is a *temporal database*, which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

## Unit - 6

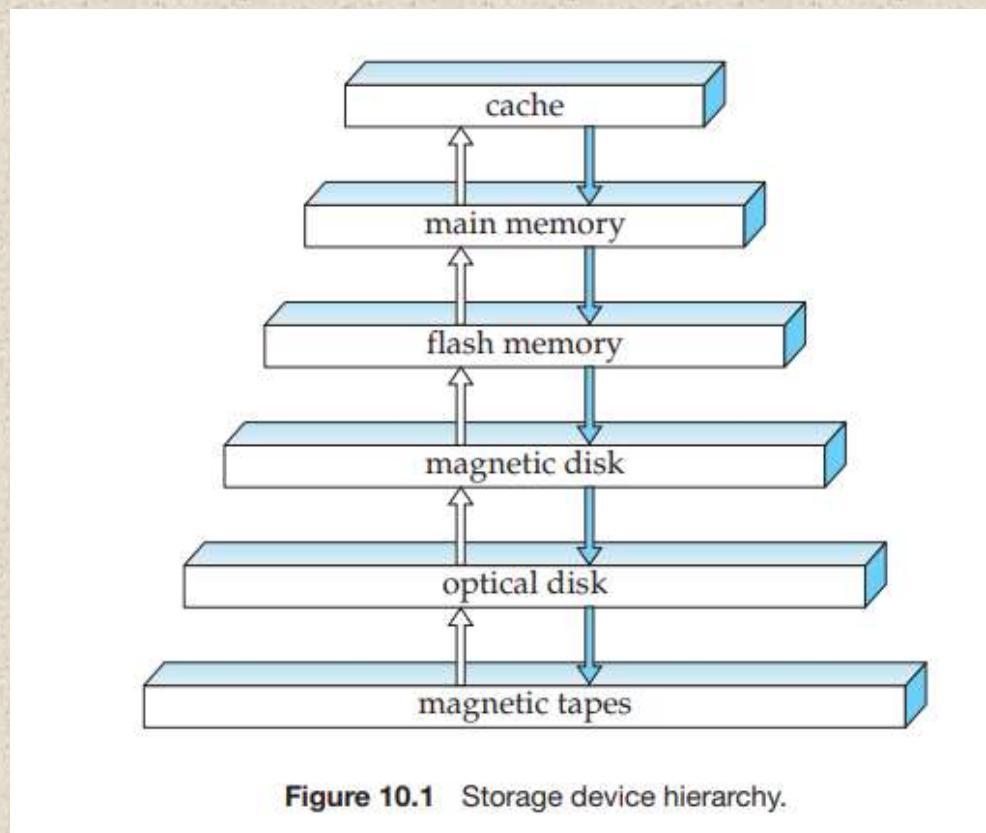
# Trends in Database Technology

# Physical Storage Media

- ◆ Although a database system provides a high-level view of data, ultimately data have to be stored as bits on one or more storage devices.
- ◆ A vast majority of databases today store data on magnetic disk and fetch data into main memory for processing, or copy data onto tapes and other backup devices for archival storage.
- ◆ The physical characteristics of storage devices play a major role in the way data are stored, in particular because access to a random piece of data on disk is much slower than memory access: Disk access takes tens of milliseconds, whereas memory access takes a tenth of a microsecond.

# Physical Storage Media

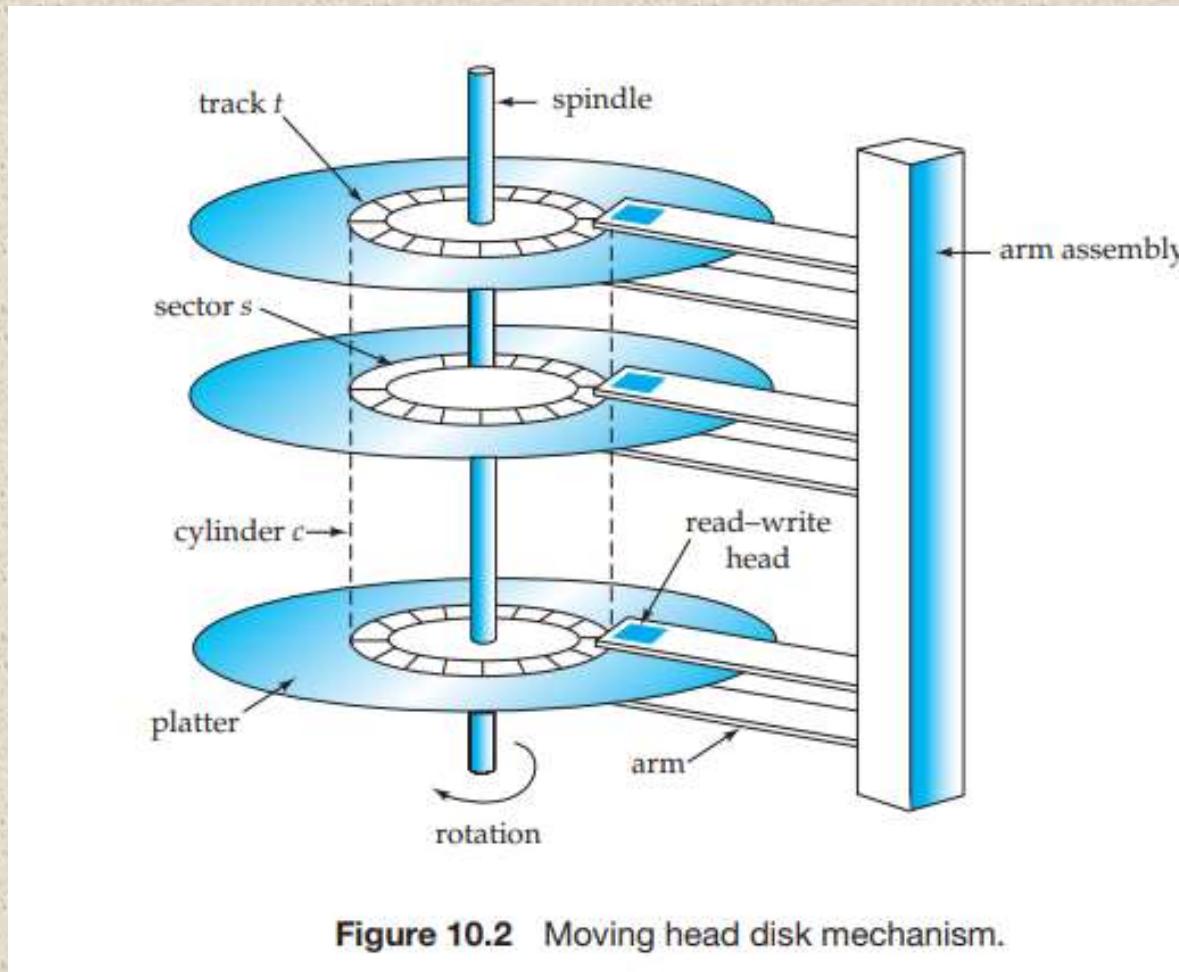
- ◆ Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability. Among the media typically available are these are cache, main memory, magnetic disk, optical disk, tape.



# Physical Storage Media

- ◆ Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability.
- ◆ Among the media typically available are these are cache, main memory, magnetic disk, optical disk, tape.

# Physical Storage Media



**Figure 10.2** Moving head disk mechanism.

# Physical Storage Media

- ◆ **Performance Measures of Disk**
- ◆ **Access time** is the time from when a read or write request is issued to when data transfer begins. To access (that is, to read or write) data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called the seek time, and it increases with the distance that the arm must move. Typical seek times range from 2 to 30 milliseconds, depending on how far the track is from the initial arm position. Smaller disks tend to have lower seek times since the head has to travel a smaller distance.
- ◆ The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests. I

# Physical Storage Media

- ◆ **Performance Measures of Disk**
- ◆ Once the head has reached the desired track, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**.
- ◆ The access time is then the sum of the seek time and the latency, and ranges from 8 to 20 milliseconds. Once the first sector of the data to be accessed has come under the head, data transfer begins.
- ◆ The data-transfer rate is the rate at which data can be retrieved from or stored to the disk.
- ◆ The final commonly used measure of a disk is the **mean time to failure (MTTF)**, which is a measure of the reliability of the disk. The mean time to failure of a disk (or of any other system) is the amount of time that, on average, we can expect the system to run continuously without any failure

# Physical Storage Media

- ◆ **Optimization of Disk-Block Access**
- ◆ **Buffering.** Blocks that are read from disk are stored temporarily in an in-memory buffer, to satisfy future requests. Buffering is done by both the operating system and the database system.
- ◆ **Read-ahead.** When a disk block is accessed, consecutive blocks from the same track are read into an in-memory buffer even if there is no pending request for the blocks. In the case of sequential access, such read-ahead ensures that many blocks are already in memory when they are requested, and minimizes the time wasted in disk seeks and rotational latency per block read. Operating systems also routinely perform read-ahead for consecutive blocks of an operating system file.

# Physical Storage Media

- ◆ Optimization of Disk-Block Access
- ◆ Scheduling. If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting the blocks in the order in which they will pass under the heads. If the desired blocks are on different cylinders, it is advantageous to request the blocks in an order that minimizes disk-arm movement. Disk-arm-scheduling algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed.
- ◆ File organization. To reduce block-access time, we can organize blocks on disk in a way that corresponds closely to the way we expect data to be accessed. For example, if we expect a file to be accessed sequentially, then we should ideally keep all the blocks of the file sequentially on adjacent cylinders.

# RAID

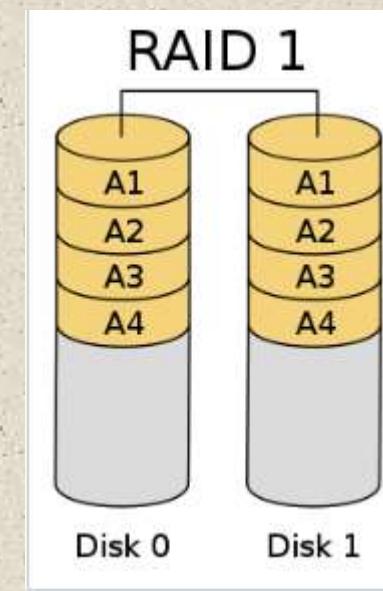
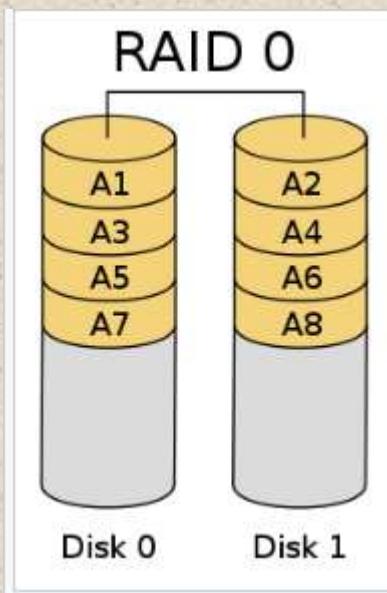
- ◆ The data-storage requirements of some applications (in particular Web, database, and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.
- ◆ Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.
- ◆ A variety of disk-organization techniques, collectively called redundant arrays of independent disks (RAID), have been proposed to achieve improved performance and reliability.

# RAID

- ◆ The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, shadowing). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is repaired.
- ◆ With multiple disks, we can improve the transfer rate as well (or instead) by **striping data** across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**. For example, if we have an array of eight disks, we write bit  $i$  of each byte to disk  $i$ .
- ◆ **Block-level striping** stripes blocks across multiple disks.

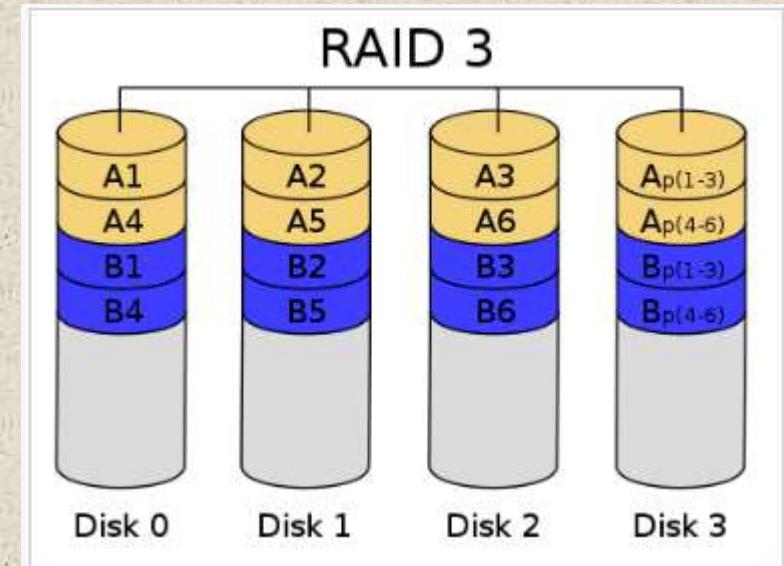
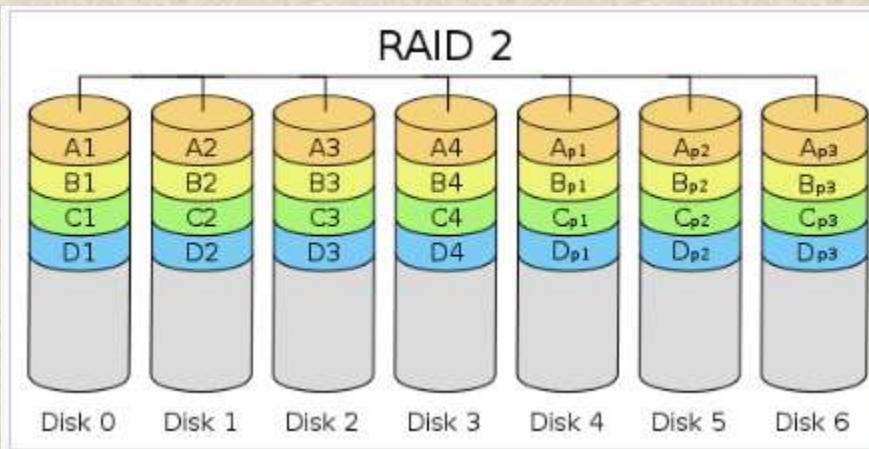
# RAID Levels

- ◆ **RAID 0** (also known as a *stripe set* or *striped volume*) splits ("stripes") data evenly across two or more disks, without parity information, redundancy.
- ◆ **RAID 1** consists of an exact copy (or *mirror*) of a set of data on two or more disks.



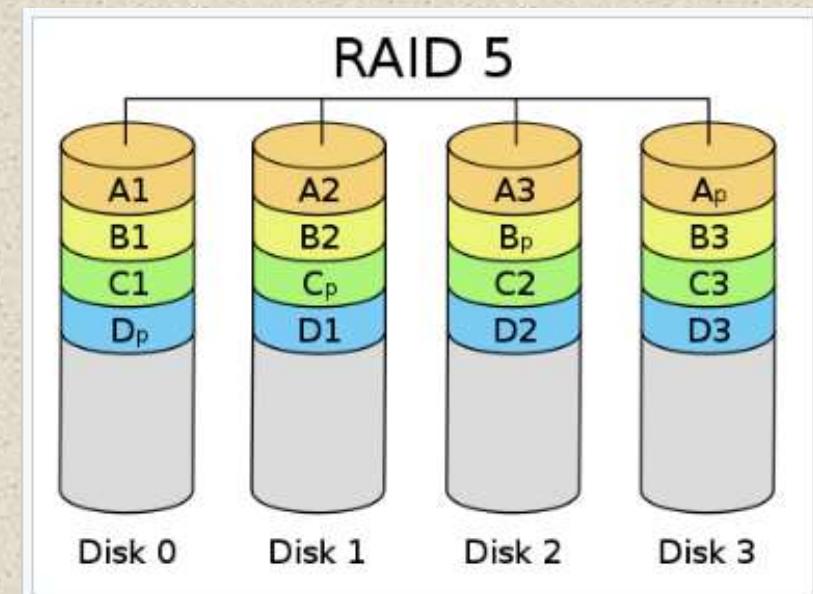
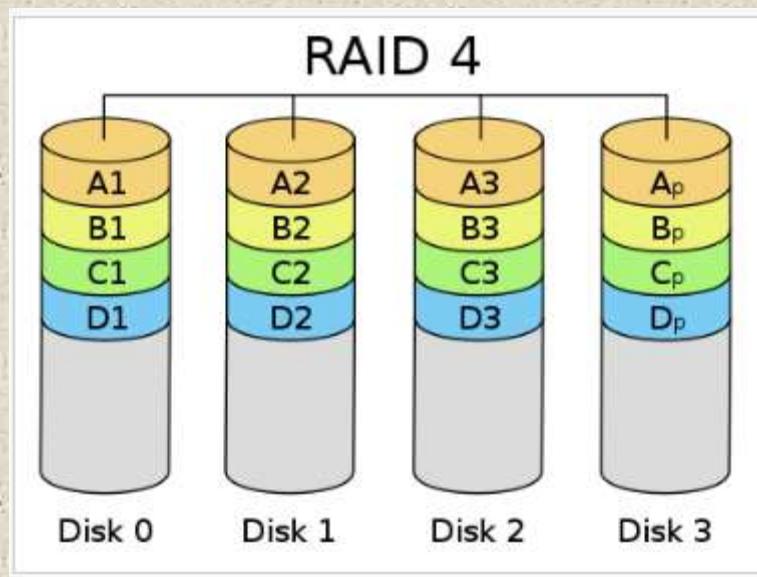
# RAID Levels

- ◆ **RAID 2**, stripes data at the bit (rather than block) level, and uses a Hamming code for error correction.
- ◆ **RAID 3**, consists of byte-level (block-level) striping with a dedicated parity disk.



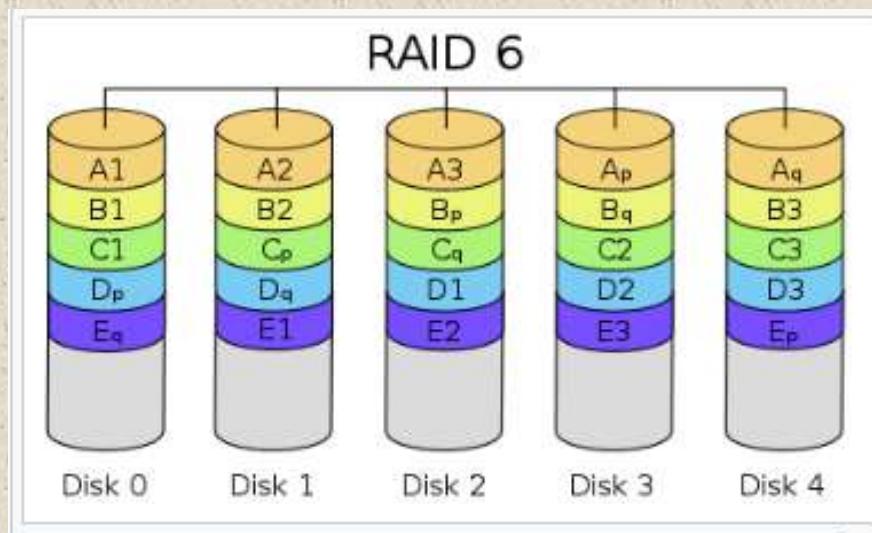
# RAID Levels

- ◆ **RAID 4** consists of block-level striping with a dedicated parity disk.
- ◆ **RAID 5** consists of block-level striping with distributed parity. Unlike in RAID 4, parity information is distributed among the drives.



# RAID Levels

- ◆ **RAID 6** extends RAID 5 by adding another parity block; thus, it uses block-level striping with two parity blocks distributed across all member disks



# Tertiary Storage

- ◆ In a large database system, some of the data may have to reside on tertiary storage. The two most common tertiary storage media are **optical disks and magnetic tapes**.
- ◆ Tapes are used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another. Tapes are also used for storing large volumes of data, such as video or image data, that either do not need to be accessible quickly or are so voluminous that magnetic-disk storage would be too expensive

# File Organization

- ◆ A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks. A file is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying file system. We need to consider ways of representing logical data models in terms of files.
- ◆ Each file is also logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.

# File Organization

- ◆ A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks. A file is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying file system. We need to consider ways of representing logical data models in terms of files.
- ◆ Each file is also logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.

# File Organization

- ◆ A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used. Its an assumption that no record is larger than a block.
- ◆ we shall require that each record is entirely contained in a single block; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.

# File Organization

- ◆ A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used. Its an assumption that no record is larger than a block.
- ◆ we shall require that each record is entirely contained in a single block; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.
- ◆ In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed-length records are easier to implement than are files of variable-length records.

# File Organization

- ◆ let us consider a file of instructor records for a university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
 ID varchar (5);
 name varchar(20);
 dept name varchar (20);
 salary numeric (8,2);
end
```

# File Organization

- ◆ Fixed Length Records:
- ◆ Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes.
- ◆ Suppose that instead of allocating a variable amount of bytes for the attributes ID, name, and dept name, we allocate the maximum number of bytes that each attribute can hold. Then, the instructor record is 53 bytes long.
- ◆ A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on.

# File Organization

- ◆ Fixed Length Records:

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 3  | 22222 | Einstein   | Physics    | 95000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |

**Figure 10.4** File containing *instructor* records.

# File Organization

- ◆ Fixed Length Records:
- ◆ Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
- ◆ It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

# File Organization

- ◆ Fixed Length Records:
- ◆ To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.
- ◆ When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead.
- ◆ Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record. It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses.

# File Organization

- ◆ Fixed Length Records:

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |

**Figure 10.5** File of Figure 10.4, with record 3 deleted and all records moved.

# File Organization

- ◆ Fixed Length Records:

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |

**Figure 10.6** File of Figure 10.4, with record 3 deleted and final record moved.

# File Organization

- ◆ Fixed Length Records:
- ◆ Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.
- ◆ A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.
- ◆ At the beginning of the file, we allocate a certain number of bytes as a file header. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as pointers, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a free list.

# File Organization

- ◆ Fixed Length Records:

|           | header |            |            |       |
|-----------|--------|------------|------------|-------|
| record 0  | 10101  | Srinivasan | Comp. Sci. | 65000 |
| record 1  |        |            |            |       |
| record 2  | 15151  | Mozart     | Music      | 40000 |
| record 3  | 22222  | Einstein   | Physics    | 95000 |
| record 4  |        |            |            |       |
| record 5  | 33456  | Gold       | Physics    | 87000 |
| record 6  |        |            |            |       |
| record 7  | 58583  | Califieri  | History    | 62000 |
| record 8  | 76543  | Singh      | Finance    | 80000 |
| record 9  | 76766  | Crick      | Biology    | 72000 |
| record 10 | 83821  | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345  | Kim        | Elec. Eng. | 80000 |

**Figure 10.7** File of Figure 10.4, with free list after deletion of records 1, 4, and 6.

# File Organization

- ◆ Fixed Length Records:
- ◆ On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.
- ◆ Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

# File Organization

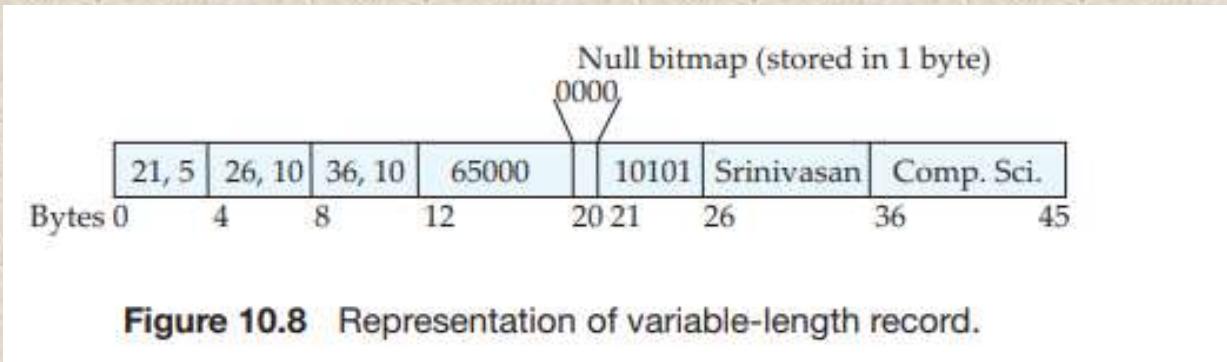
- ◆ Variable Length Records:
- ◆ Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields, such as arrays or multisets.
- ◆ Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:
  - How to represent a single record in such a way that individual attributes can be extracted easily.
  - How to store variable-length records within a block, such that records in a block can be extracted easily.

# File Organization

- ◆ Variable Length Records:
- ◆ The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes.
- ◆ Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value.
- ◆ Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (offset, length), where offset denotes where the data for that attribute begins within the record, and length is the length in bytes of the variable-sized attribute. The values for these attributes are stored consecutively, after the initial fixed-length part of the record.
- ◆ Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length.

# File Organization

- ◆ Variable Length Records:



**Figure 10.8** Representation of variable-length record.

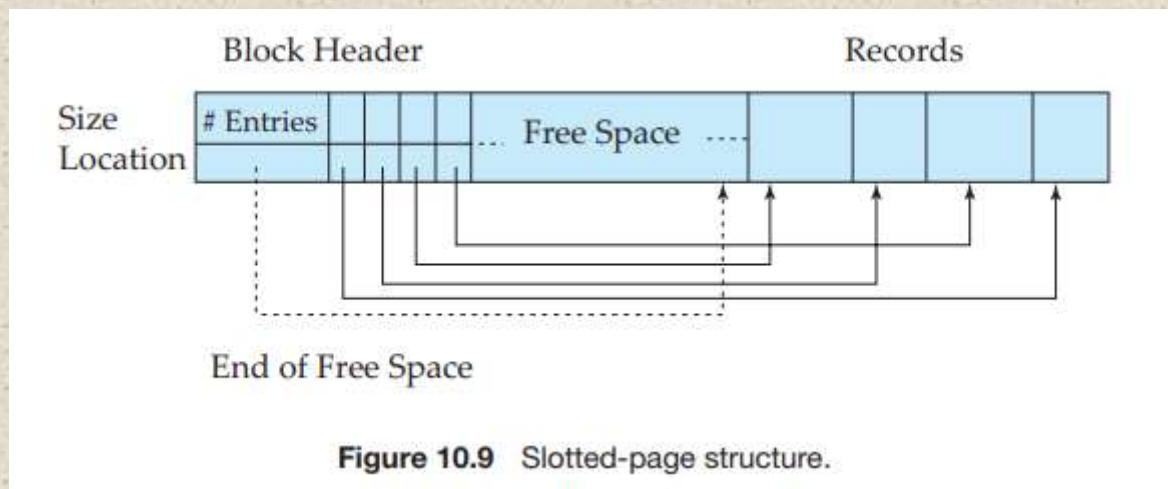
- ◆ An example of such a record representation is shown in Figure 10.8. The figure shows an instructor record, whose first three attributes ID, name, and dept name are variable-length strings, and whose fourth attribute salary is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The salary attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

# File Organization

- ◆ Variable Length Records:
- ◆ The Figure 10.8 also illustrates the use of a null bitmap, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored.
- ◆ In some representations, the null bitmap is stored at the beginning of the record, and for attributes that are null, no data (value, or offset/length) are stored at all. Such a representation would save some storage space.

# File Organization

- ◆ Variable Length Records:
- ◆ The slotted-page structure is commonly used for organizing records within a block, and is shown in Figure 10.9.
- ◆ There is a header at the beginning of each block, containing the following information:
  - The number of record entries in the header.
  - The end of free space in the block.
  - An array whose entries contain the location and size of each record



# File Organization

- ◆ Variable Length Records:
- ◆ The actual records are allocated contiguously in the block, starting from the end of the block.
- ◆ The free space in the block is contiguous, between the final entry in the header array, and the first record.
- ◆ If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

# Organization of Records in Files

- ◆ **Heap file organization:** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation
- ◆ **Sequential file organization:** Records are stored in sequential order, according to the value of a “search key” of each record.
- ◆ **Hashing file organization:** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed

## Data-Dictionary Storage

- ◆ A relational database system needs to maintain data about the relations, such as the schema of the relations. In general, such “data about data” is referred to as metadata.
- ◆ Relational schemas and other metadata about relations are stored in a structure called the **data dictionary or system catalog**.
- ◆ Among the types of information that the system must store are these:
  - Names of the relations.
  - Names of the attributes of each relation.
  - Domains and lengths of attributes.
  - Names of views defined on the database, and definitions of those views.
  - Integrity constraints (for example, key constraints)

# Data-Dictionary Storage

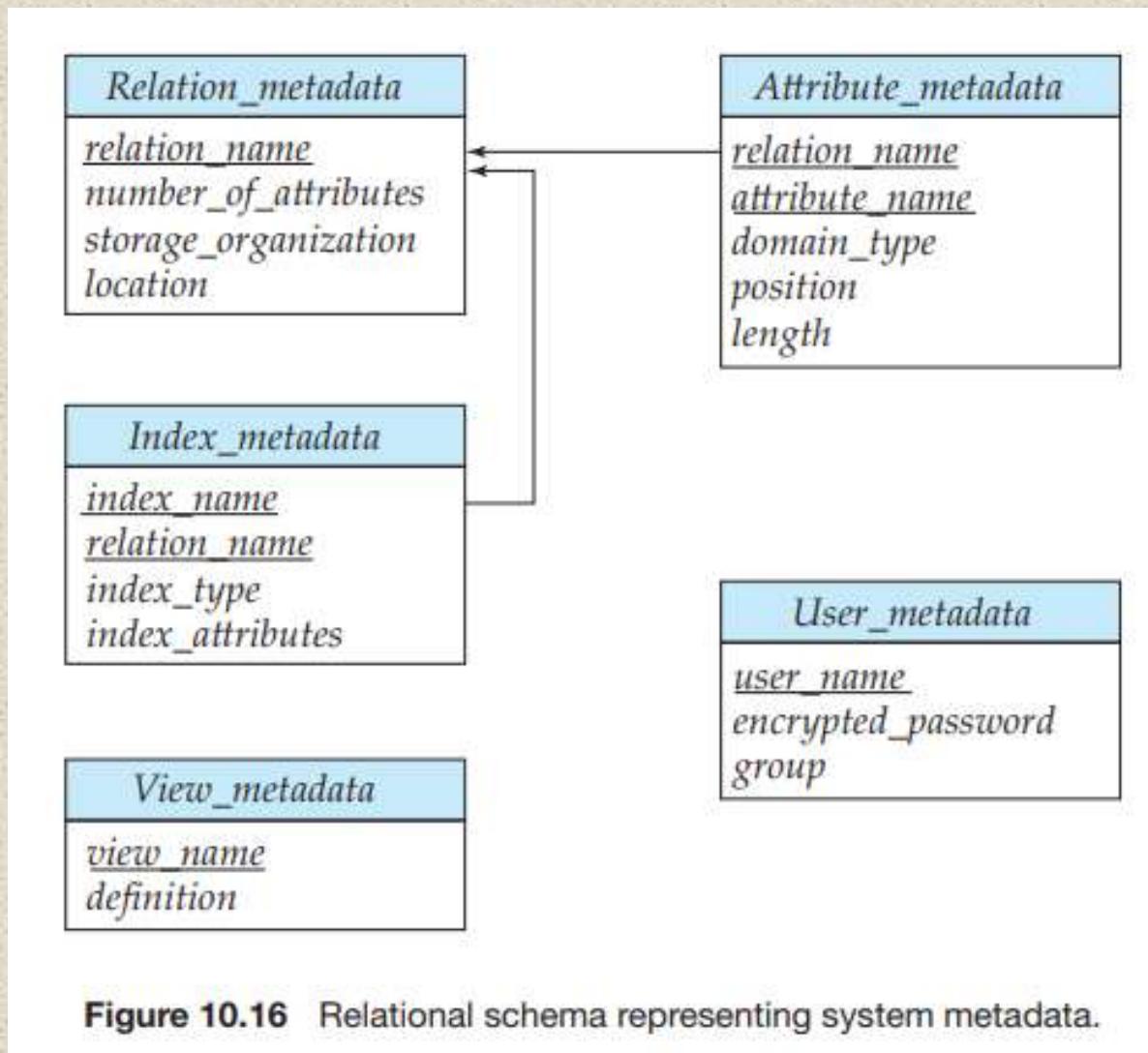


Figure 10.16 Relational schema representing system metadata.

# Basic Concepts of Indexing

- ◆ An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.
- ◆ Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

# Basic Concepts of Indexing

- ◆ Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
- ◆ An **index** is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. An **index allows** us to efficiently retrieve all records that satisfy search conditions on the search key fields of the index.
- ◆ We can also create additional indexes on a given collection of data records, each with a different search key, to speed up search operations that are not efficiently supported by the file organization used to store the data records.

# Basic Concepts of Indexing

- ◆ There are two basic kinds of indices:
  - **Ordered indices: Based on a sorted ordering of the values.**
  - **Hash indices. Based on a uniform distribution of values across a range of buckets.** The bucket to which a value is assigned is determined by a function, called a *hash function*.

# Basic Concepts of Indexing

- ◆ Each technique must be evaluated on the basis of these factors:
  - **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
  - **Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.
  - **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
  - **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
  - **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

# Ordered Indices

- ◆ An attribute or set of attributes used to look up records in a file is called a search key.
- ◆ To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, **an ordered index stores** the values of the search keys in sorted order, and associates with each search key the records that contain it.
- ◆ A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index is an index** whose search key also defines the sequential order of the file.

# Ordered Indices

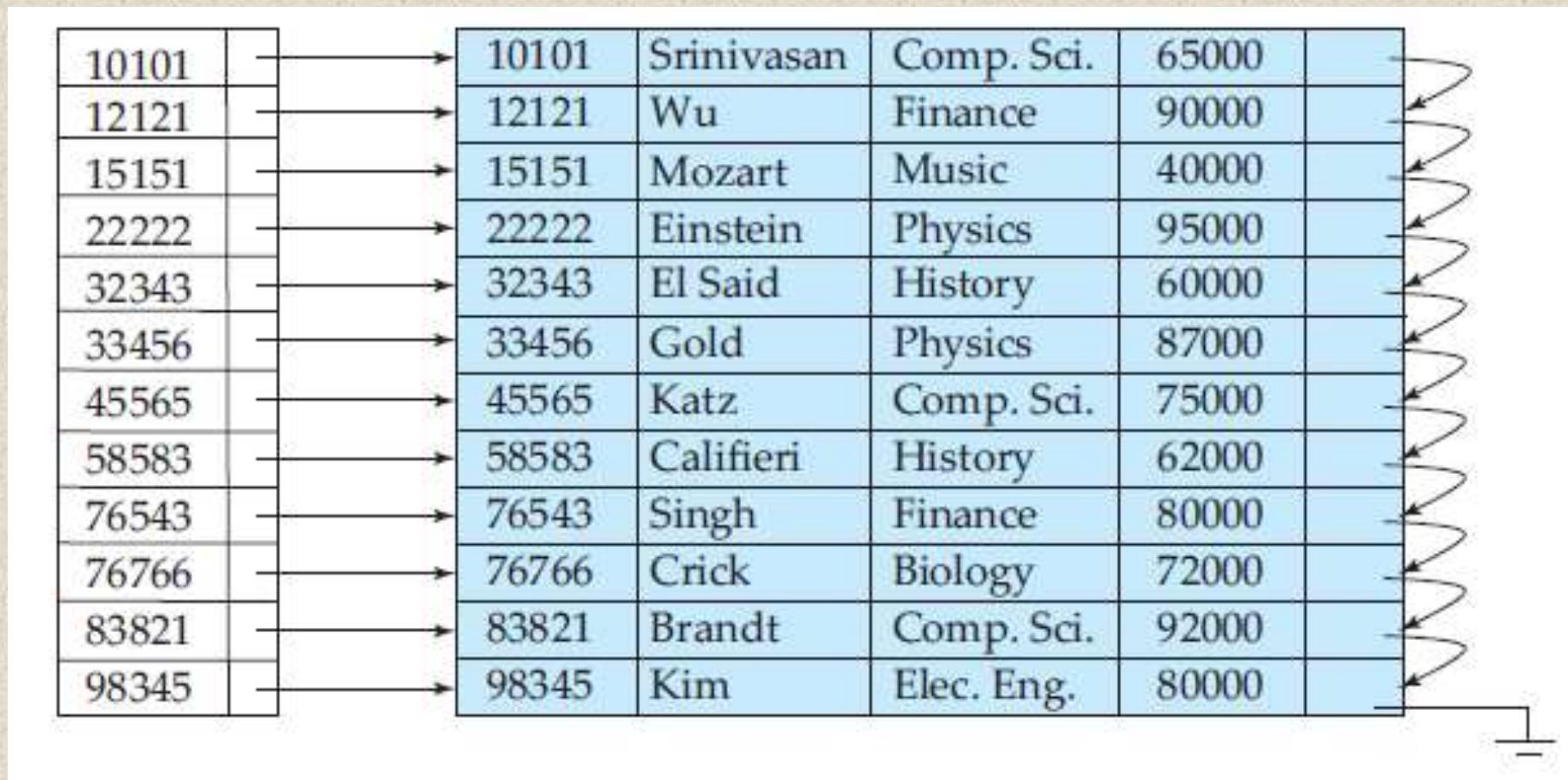
- ◆ **Clustering indices** are also called **primary indices**; the term **primary index may appear to denote an index on a primary key**, but such indices can in fact be built on any search key.
- ◆ The search key of a clustering index is often the primary key, although that is not necessarily so.
- ◆ Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices, or secondary indices**.
- ◆ An **index entry, or index record, consists of a search-key value and pointers to** one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

# Ordered Indices

- ◆ There are two types of ordered indices; **Dense and Sparse**.
- ◆ **Dense index:** In a dense index, an index entry appears for every search-key value in the file.
- ◆ In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value.
- ◆ The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key.

# Ordered Indices

- ◆ Dense Index:

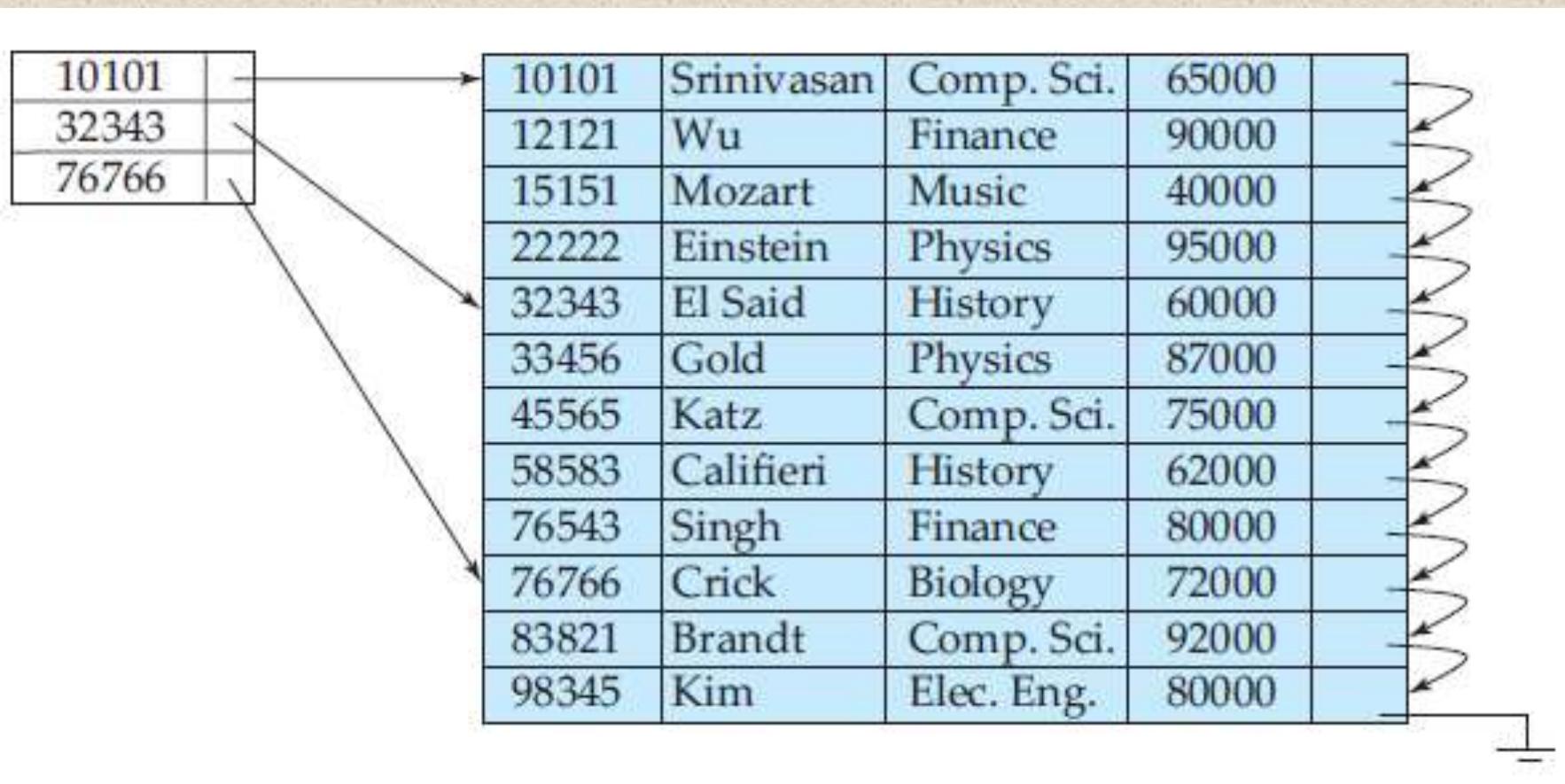


# Ordered Indices

- ◆ **Sparse index:**
- ◆ In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value.
- ◆ To locate a record, we find the index entry with the **largest search-key value** that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

# Ordered Indices

- ◆ Sparse index:



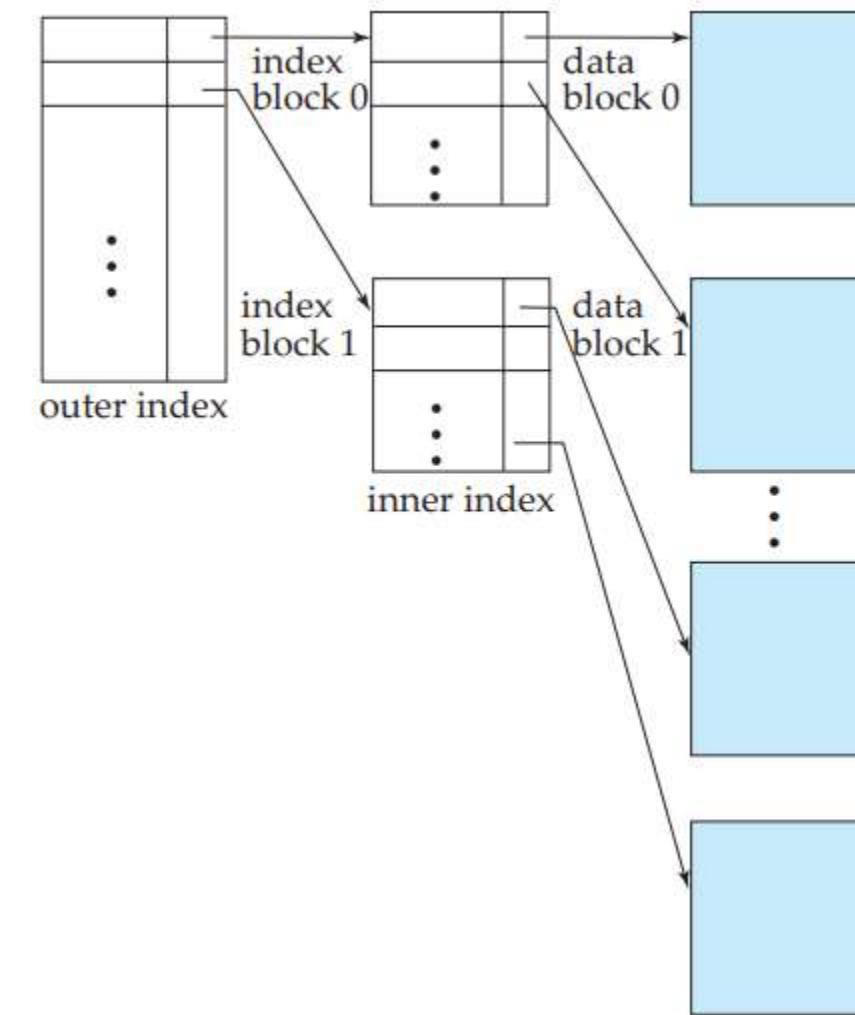
# Ordered Indices

- ◆ It is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

# Multilevel Indices

- ◆ Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.
- ◆ Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.
- ◆ Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing.

# Multilevel Indices



# B+-Tree Index Files

- ◆ The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.
- ◆ The B+-tree index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B+-tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between  $n/2$  and  $n$  children, where  $n$  is fixed for a particular tree.

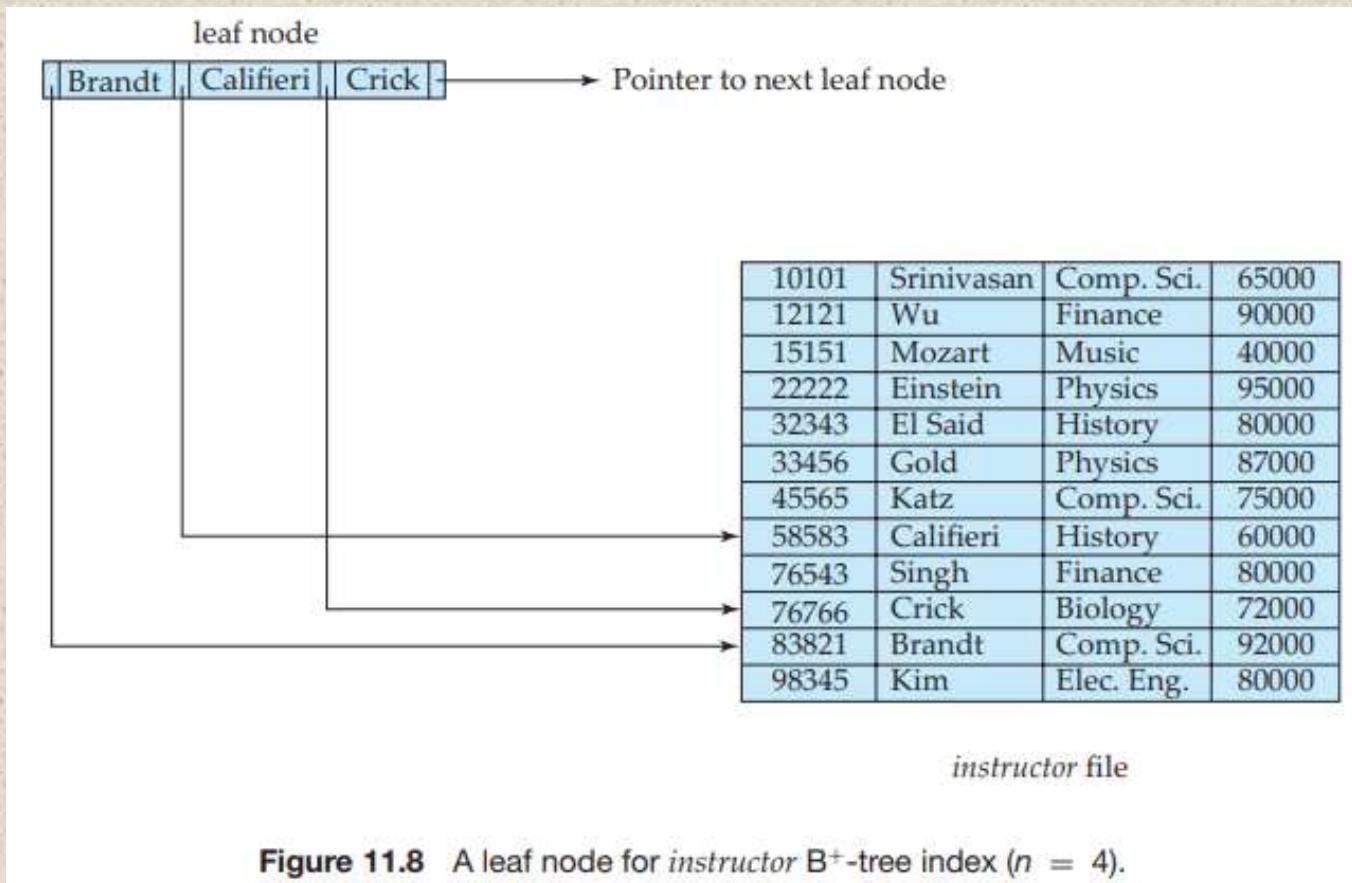
# B+-Tree Index Files

- ◆ A B+-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. A node of a B+- tree contains up to  $n - 1$  search-key values  $K_1, K_2, \dots, K_{n-1}$ , and  $n$  pointers  $P_1, P_2, \dots, P_n$ .
- ◆ The search-key values within a node are kept in sorted order; thus, if  $i < j$ , then  $K_i < K_j$ . We consider first the structure of the leaf nodes. For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  points to a file record with search-key value  $K_i$ .

|       |       |       |         |           |           |       |
|-------|-------|-------|---------|-----------|-----------|-------|
| $P_1$ | $K_1$ | $P_2$ | $\dots$ | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|-------|-------|-------|---------|-----------|-----------|-------|

**Figure 11.7** Typical node of a  $B^+$ -tree.

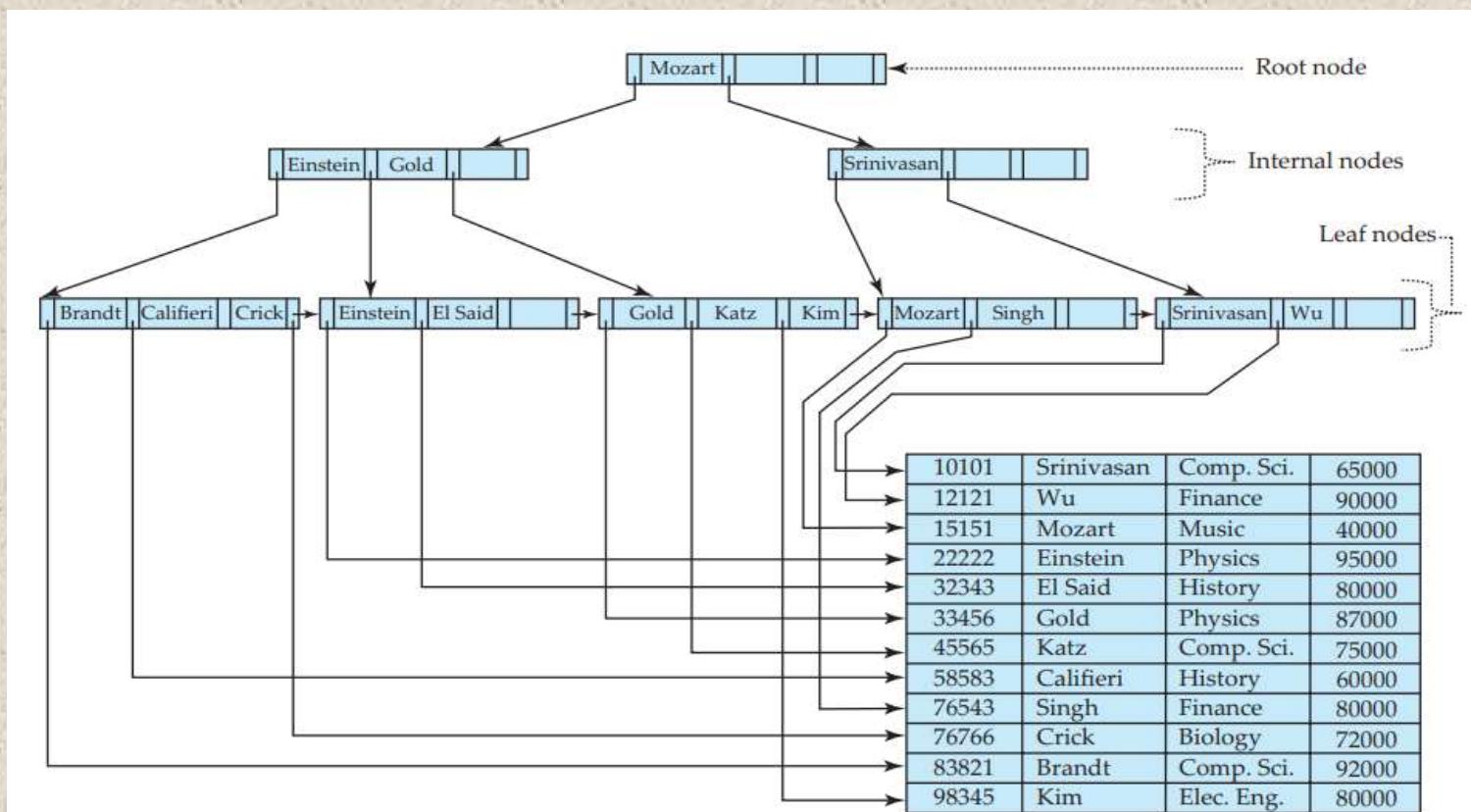
# B+-Tree Index Files



**Figure 11.8** A leaf node for *instructor* B+-tree index ( $n = 4$ ).

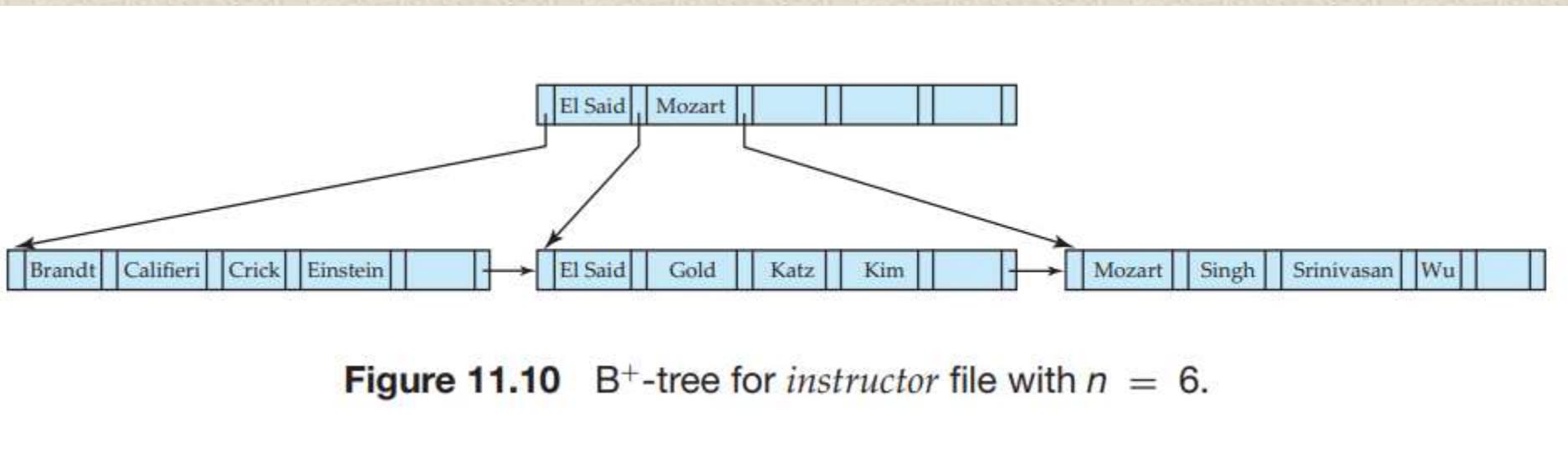
# B+-Tree Index Files

- The nonleaf or internal nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes.



# B+-Tree Index Files

- ◆ The nonleaf nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes.



**Figure 11.10** B+-tree for *instructor* file with  $n = 6$ .

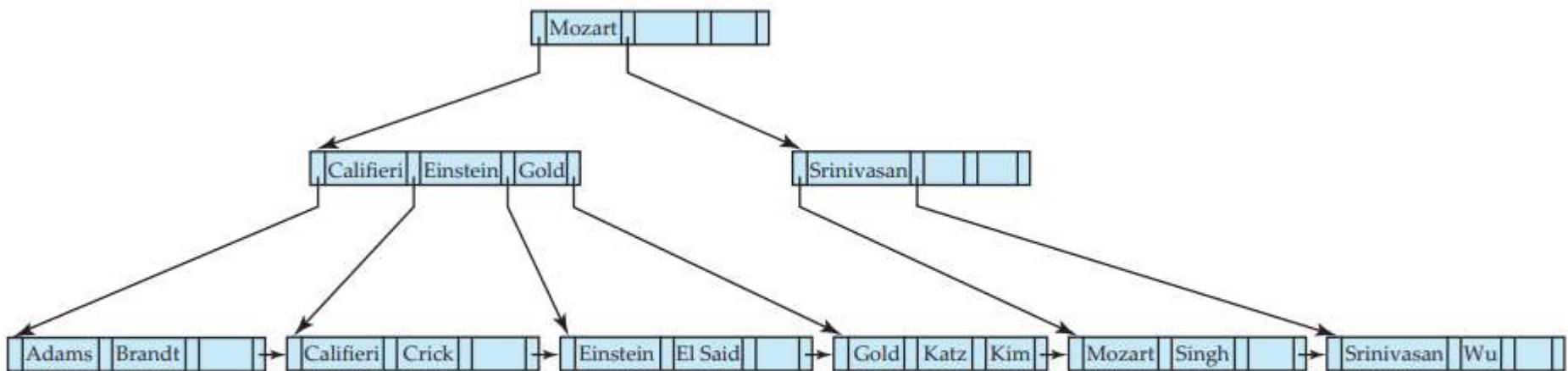
# B+-Tree Index Files

- ◆ We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the instructor relation, with the name value being Adams. We then need to insert an entry for “Adams” into the B+-tree of Figure 11.9.
- ◆ Using the algorithm for lookup, we find that “Adams” should appear in the leaf node containing “Brandt”, “Califieri”, and “Crick.” There is no room in this leaf to insert the search-key value “Adams.” Therefore, the node is split into two nodes. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on inserting “Adams”. The search-key values “Adams” and “Brandt” are in one leaf, and “Califieri” and “Crick” are in the other.

# B+-Tree Index Files

- ◆ Having split a leaf node, we must insert the new leaf node into the B+-tree structure. In our example, the new node has “Califieri” as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split.
- ◆ The B+-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

# B+-Tree Index Files



**Figure 11.13** Insertion of “Adams” into the B<sup>+</sup>-tree of Figure 11.9.

# B+-Tree Index Files

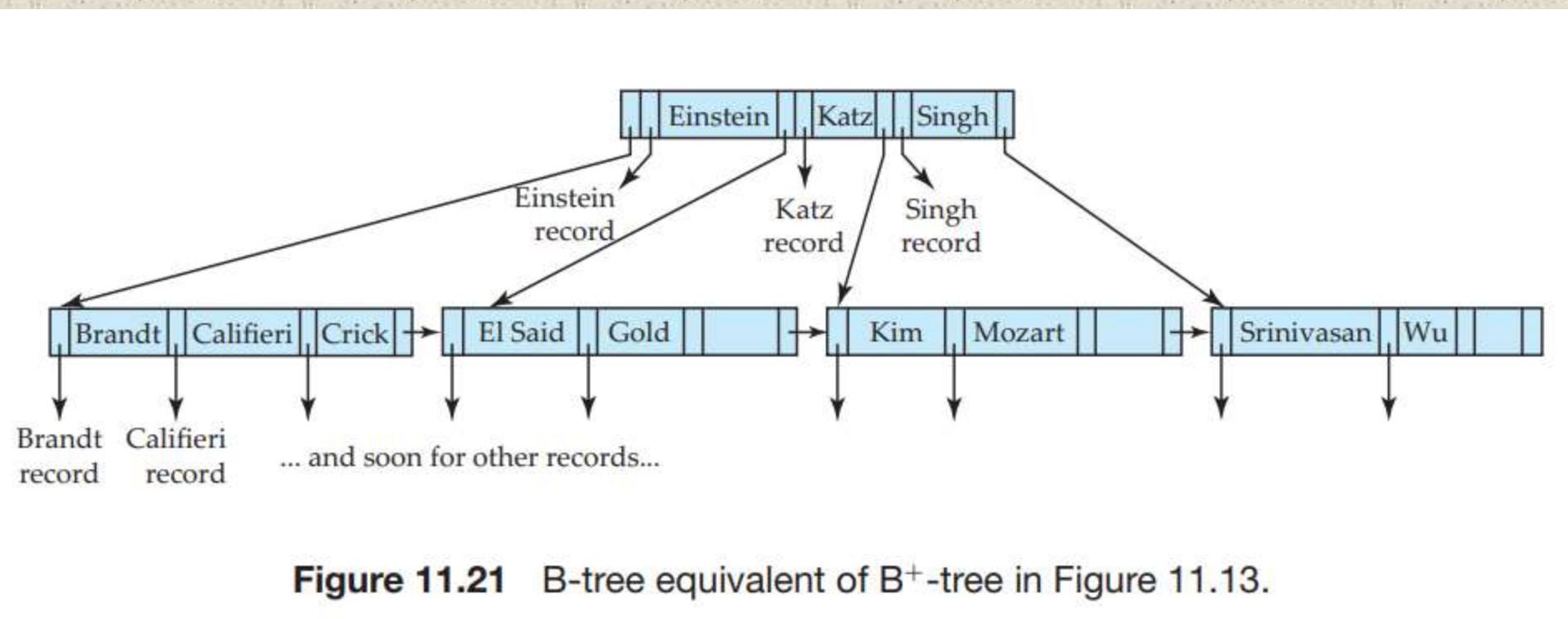
- ◆ Having split a leaf node, we must insert the new leaf node into the B+-tree structure. In our example, the new node has “Califieri” as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split.
- ◆ The B+-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

# B-Tree Index Files

- ◆ B-tree indices are similar to B+-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the B+-tree of Figure 11.13, the search keys “Califieri”, “Einstein”, “Gold”, “Mozart”, and “Srinivasan” appear in nonleaf nodes, in addition to appearing in the leaf nodes. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.
- ◆ A B-tree allows search-key values to appear only once (if they are unique), unlike a B+-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node.

# B-Tree Index Files

- ◆ Figure 11.21 shows a B-tree that represents the same search keys as the B+-tree of Figure 11.13. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B+-tree index.



**Figure 11.21** B-tree equivalent of B<sup>+</sup>-tree in Figure 11.13.

# Multiple-Key Access

- ◆ We can use multiple indices if they exist, or to use an index built on a multiatribute search key.

- ◆ **Using Multiple Single-Key Indices**

- ◆ Assume that the *instructor* file has two indices: one for *dept name* and one for *salary*. Consider the following query: “Find all instructors in the Finance department with salary equal to \$80,000.”

We write

```
select ID
from instructor
where dept name = 'Finance' and salary= 80000;
```

- ◆ There are three strategies possible for processing this query:

# Multiple-Key Access

- ◆ **Using Multiple Single-Key Indices**
- ◆ There are three strategies possible for processing the above query:
- ◆ **Use the index on *dept name* to find all records pertaining to the *Finance* department.** Examine each such record to see whether *salary*= 80000.
- ◆ **Use the index on *salary* to find all records pertaining to instructors with** salary of \$80,000. Examine each such record to see whether the department name is “Finance”.
- ◆ **Use the index on *dept name* to find pointers to all records pertaining to the *Finance* department.** Also, **use the index on *salary* to find pointers to all records pertaining to instructors with a salary of \$80,000.** Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to instructors of the Finance department and with salary of \$80,000.

# Multiple-Key Access

- ◆ **Indices on Multiple Keys**
- ◆ An alternative strategy for this case is to create and use an index on a composite search key (*dept name, salary*)—*that is, the search key consisting of the department name concatenated with the instructor salary.*
- ◆ We can use an ordered (B+-tree) index on the above composite search key to answer efficiently queries of the form

**select *ID***

**from *instructor***

**where *dept name* = 'Finance' and *salary*= 80000**

# Creating Index

- ◆ We can create index in sql like;
- ◆ CREATE INDEX *index\_name*  
ON *table\_name* (*column1*, *column2*, ...);
- ◆ For Example;
- ◆ The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
 ON Persons (LastName);
```

- ◆ If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

```
CREATE INDEX idx_pname
 ON Persons (LastName, FirstName);
```

# Droping Index

- ◆ The DROP INDEX statement is used to delete an index in a table.  
There are various way to drop index as;

```
ALTER TABLE table_name
DROP INDEX index_name;
```

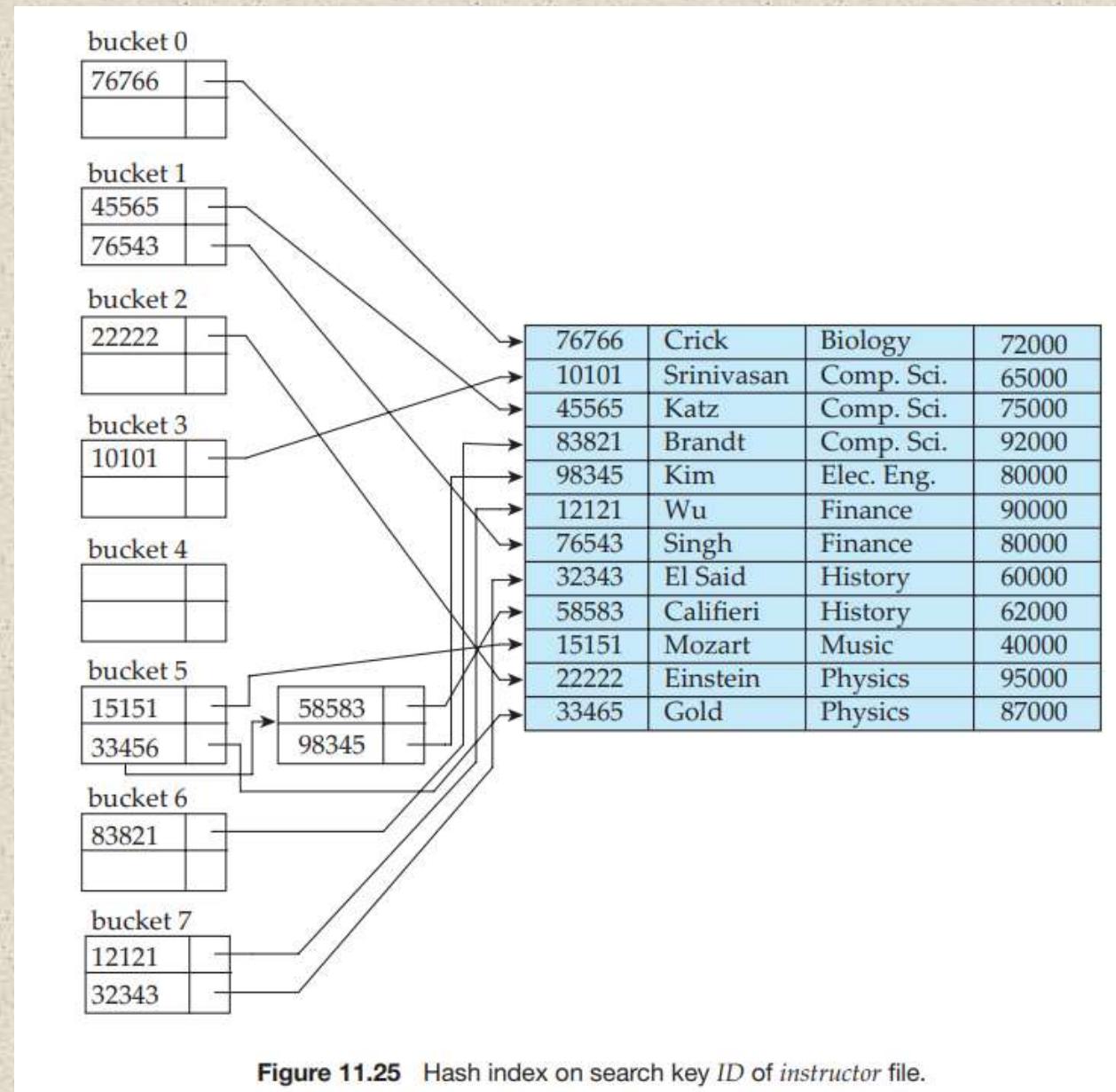
# Hashing

- ◆ One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of hashing allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices.
- ◆ In hashing, the term bucket is used to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.
- ◆ Formally, let  $K$  denote the set of all search-key values, and let  $B$  denote the set of all bucket addresses. A hash function  $h$  is a function from  $K$  to  $B$ . Let  $h$  denote a hash function.

# Hashing

- ◆ To insert a record with search key  $K_i$ , we compute  $h(K_i)$ , which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.
- ◆ To perform a lookup on a search-key value  $K_i$ , we simply compute  $h(K_i)$ , then search the bucket with that address.

# Hashing



# Static Hashing Vs Dynamic Hashing

- ◆ The **main difference** between static and dynamic hashing is that, **in static hashing, the resultant data bucket address is always the same** while, **in dynamic hashing, the data buckets grow or shrink according to the increase and decrease of records.**
- ◆ In static hashing, the resultant data bucket address is always the same. In other words, the bucket address does not change. Thus, in this method, the number of data buckets in memory remains constant throughout.
- ◆ An issue in static hashing is bucket overflow. Dynamic hashing helps to overcome this issue. It is also called **Extendable hashing method**. In this method, the data buckets increase and decrease depending on the number of records.

# Static Hashing Vs Dynamic Hashing

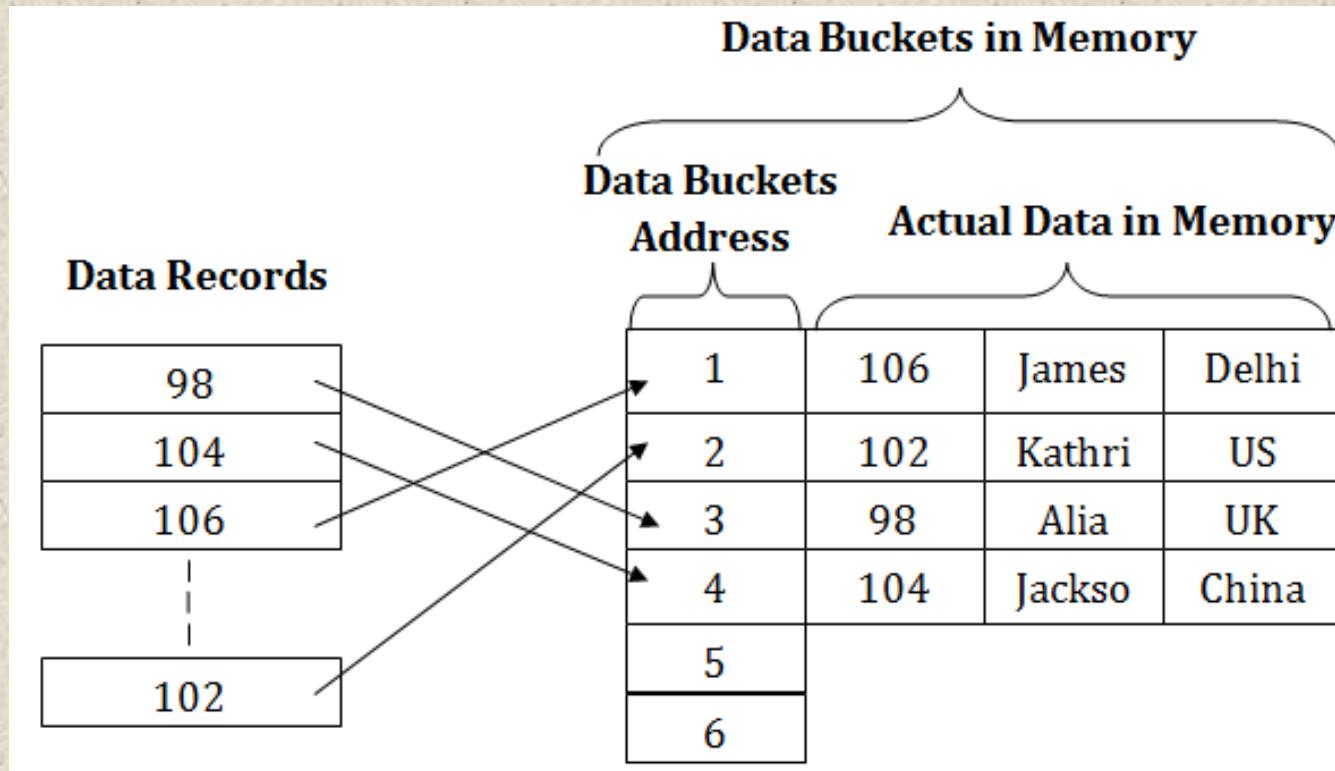
- ◆ Static hashing is suitable for applications where quick access to data is required, and the amount of data to be stored is known in advance. Dynamic hashing is suitable for applications where the amount of data to be stored is not known in advance, and scalability is important.

# Static Hashing Vs Dynamic Hashing

- ◆ In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for  $\text{EMP\_ID} = 103$  using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.
- ◆ Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.
- ◆ If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**.

# Static Hashing Vs Dynamic Hashing

- ◆ Static Hashing



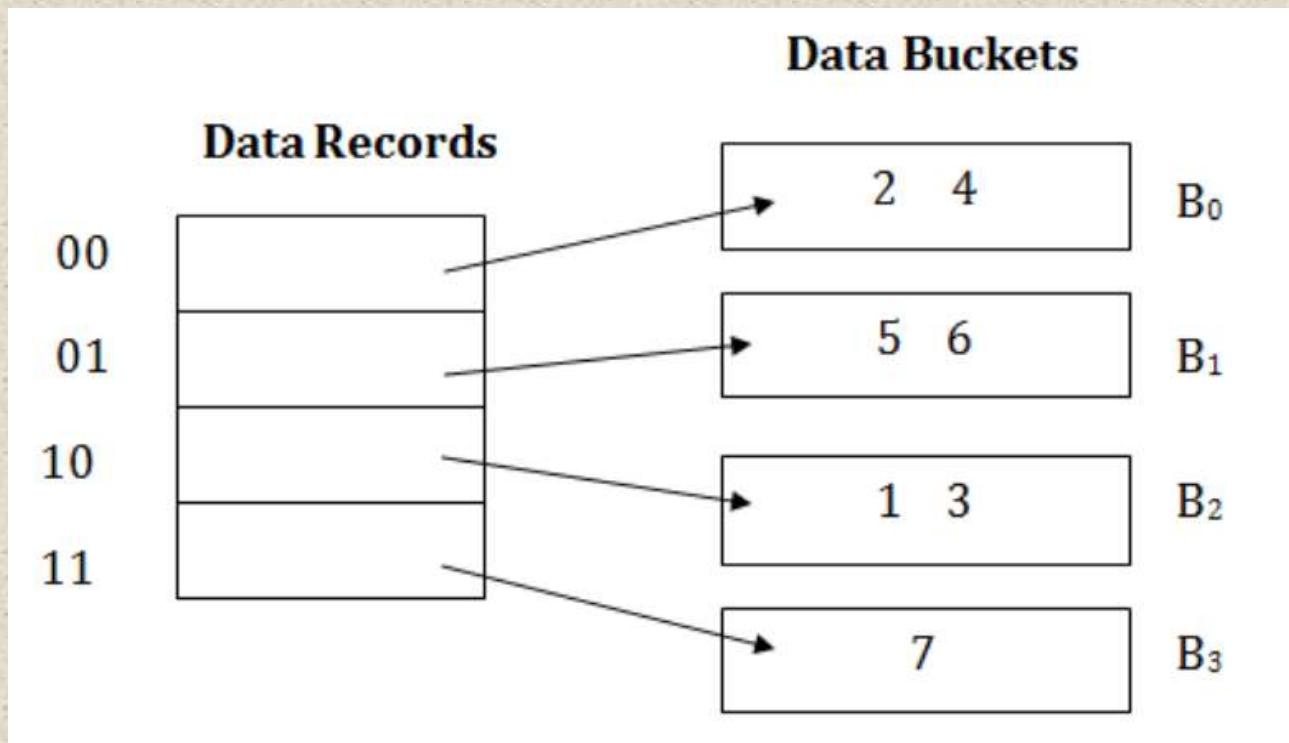
# Static Hashing Vs Dynamic Hashing

- ◆ Dynamic Hashing
- ◆ Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

| <b>Key</b> | <b>Hash address</b> |
|------------|---------------------|
| 1          | 11010               |
| 2          | 00000               |
| 3          | 11110               |
| 4          | 00000               |
| 5          | 01001               |
| 6          | 10101               |
| 7          | 10111               |

# Static Hashing Vs Dynamic Hashing

- ◆ Dynamic Hashing
- ◆ The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.

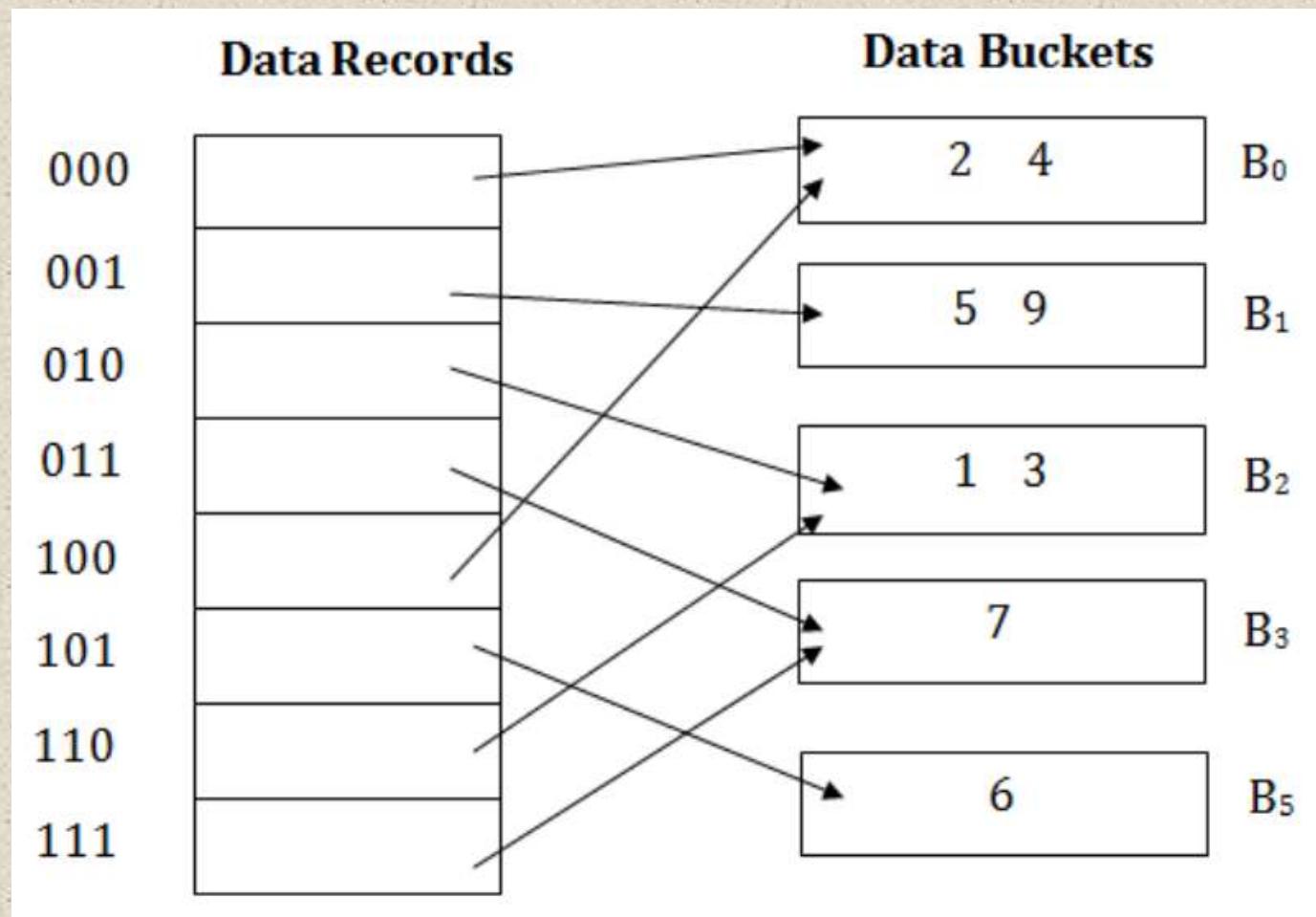


# Static Hashing Vs Dynamic Hashing

- ◆ Dynamic Hashing
- ◆ To insert key 9 with hash address 10001 into the above structure:
- ◆ Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
- ◆ The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- ◆ Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- ◆ Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
- ◆ Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.

# Static Hashing Vs Dynamic Hashing

- ◆ Dynamic Hashing

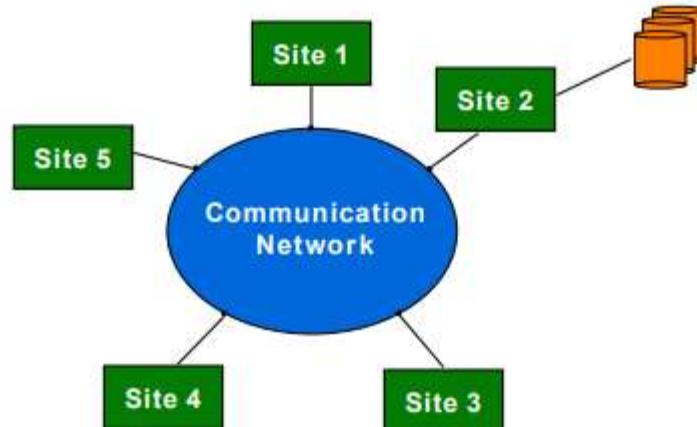


# Distributed Database

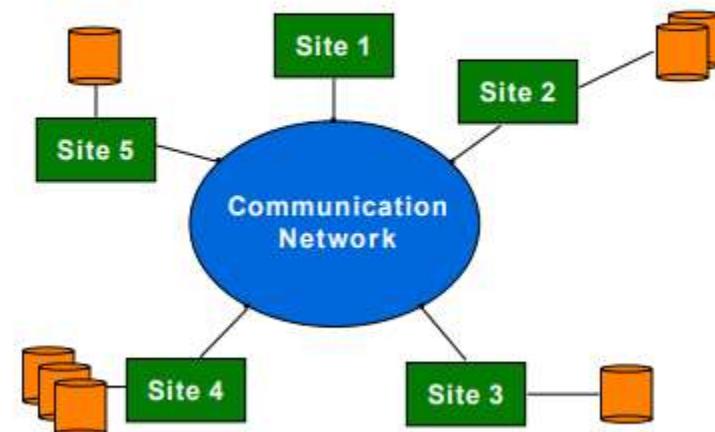
- ◆ Distributed database system is a type of database management system that stores data across multiple computers or sites that are connected by a network.
- ◆ A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network.
- ◆ A distributed database management system (D-DBMS) is the software that manages the DDB and provides an access mechanism that makes this distribution transparent to the users.

# Distributed Database

## Centralized DBMS on a Network



## Distributed DBMS Environment



# Distributed Database

- ◆ In a **homogeneous distributed database** system, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests. In such a system, local sites surrender a portion of their autonomy in terms of their right to change schemas or database-management system software. That software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites.
- ◆ In contrast, in a **heterogeneous distributed database**, different sites may use different schemas, and different database-management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing, while the divergence in software becomes a hindrance for processing transactions that access multiple sites.

# Distributed Database

- ◆ Consider a relation  $r$  that is to be stored in the database. There are two approaches to storing this relation in the distributed database:
  - Replication. The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation  $r$ .
  - Fragmentation. The system partitions the relation into several fragments, and stores each fragment at a different site.
- ◆ Fragmentation and replication can be combined: A relation can be partitioned into several fragments and there may be several replicas of each fragment.

# Distributed Database

- ◆ **Replication**
- ◆ If relation  $r$  is replicated, a copy of relation  $r$  is stored in two or more sites. In the most extreme case, we have **full replication**, in which a copy is stored in every site in the system
- ◆ In general, replication enhances the performance of read operations and increases the availability of data to read-only transactions. However, update transactions incur greater overhead.
- ◆ Controlling concurrent updates by several transactions to replicated data is more complex than in centralized systems.

# Distributed Database

- ◆ **Fragmentation**
- ◆ If relation  $r$  is fragmented,  $r$  is divided into a number of fragments  $r_1, r_2, \dots, r_n$ . These fragments contain sufficient information to allow reconstruction of the original relation  $r$ . There are two different schemes for fragmenting a relation: **horizontal fragmentation and vertical fragmentation**.
- ◆ **Horizontal fragmentation** splits the relation by assigning each tuple of  $r$  to one or more fragments.
- ◆ **Vertical fragmentation** splits the relation by decomposing the scheme  $R$  of relation  $r$ .

# Distributed Database

- ◆ **Horizontal Fragmentation**
- ◆ **In horizontal fragmentation**, a relation  $r$  is partitioned into a number of subsets,  $r_1, r_2, \dots, r_n$ . Each tuple of relation  $r$  must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.
- ◆ In general, a **horizontal fragment** can be defined as a selection on the global relation  $r$ . That is, we use a predicate  $P_i$  to construct fragment  $r_i$  :

$$r_i = P_i(r)$$

- ◆ We reconstruct the relation  $r$  by taking the union of all fragments; that is:

$$r = r_1 \cup r_2 \cup \dots \cup r_n$$

# Distributed Database

- ◆ **Horizontal Fragmentation**
- ◆ As an illustration, the account relation can be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.
- ◆ If the banking system has only two branches—Hillside and Valleyview — then there are two different fragments:
  - account 1 = branch name = “Hillside” (account)
  - account 2 = branch name = “Valleyview” (account)

# Distributed Database

- ◆ Vertical Fragmentation
- ◆ **Vertical fragmentation** of  $r(R)$  involves the definition of several subsets of attributes  $R_1, R_2, \dots, R_n$  of the schema  $R$  so that:

$$R = R_1 \cup R_2 \cup \dots \cup R_n$$

- ◆ Each fragment  $r_i$  of  $r$  is defined by:  $r_i = R_i(r)$ .
- ◆ The fragmentation should be done in such a way that we can reconstruct relation  $r$  from the fragments by taking the natural join:

$$r = r_1 \star r_2 \star r_3 \star \dots \star r_n$$

# Distributed Database

- ◆ **Vertical Fragmentation**
- ◆ One way of ensuring that the relation  $r$  can be reconstructed is to include the primary-key attributes of  $R$  in each  $R_i$ . More generally, any superkey can be used. It is often convenient to add a special attribute, called a tuple-id, to the schema  $R$ . The tuple-id value of a tuple is a unique value that distinguishes the tuple from all other tuples. The tuple-id attribute thus serves as a candidate key for the augmented schema, and is included in each  $R_i$ .
- ◆ To illustrate vertical fragmentation, consider a university database with a relation *employee info* that stores, for each employee, employee id, name, designation, and salary. For privacy reasons, this relation may be fragmented into a relation *employee private info* containing employee id and salary, and another relation *employee public info* containing attributes employee id, name, and designation. These may be stored at different sites, again, possibly for security

# Distributed Database

- ◆ **Transparency**
- ◆ The user of a distributed database system should not be required to know where the data are physically located nor how the data can be accessed at the specific local site.
- ◆ This characteristic, called **data transparency**, can take several forms:
  - **Fragmentation transparency**. Users are not required to know how a relation has been fragmented.
  - **Replication transparency**. Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.
  - **Location transparency**. Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

# Client Server Technology

- ◆ Refer Handouts from Unit 1.2 Data Models Slide No. 47 to 56.

# Multidimensional Databases

- ◆ Multidimensional databases are used mostly for OLAP (online analytical processing) and data warehousing. They can be used to show multiple dimensions of data to users .
- ◆ A multidimensional database is created from multiple relational databases. While relational databases allow users to access data in the form of queries, the multidimensional databases allow users to ask analytical questions related to business or market trends.
- ◆ The multidimensional databases uses MOLAP (multidimensional online analytical processing) to access its data. They allow the users to quickly get answers to their requests by generating and analyzing the data rather quickly.
- ◆ The data in multidimensional databases is stored in a data cube format. This means that data can be seen and understood from many dimensions and perspectives.

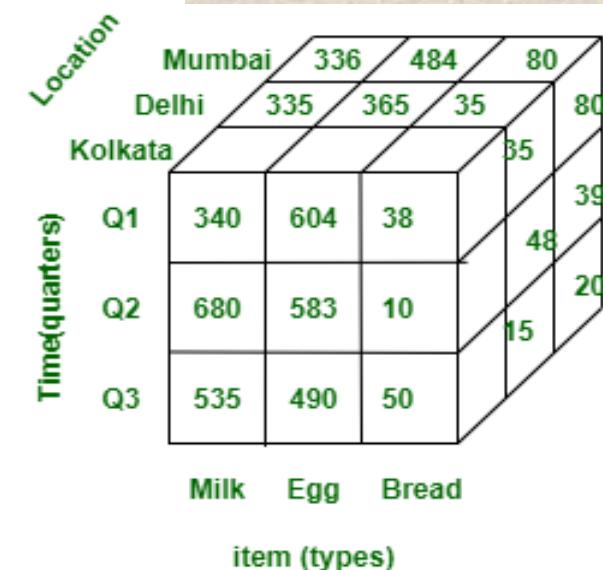
# Multidimensional Databases

- ◆ It represents data in the form of data cubes. Data cubes allow to model and view the data from many dimensions and perspectives. It is defined by dimensions and facts and is represented by a fact table. Facts are numerical measures and fact tables contain measures of the related dimensional tables or names of the facts.
- ◆ Multidimensional databases, also known as OLAP (Online Analytical Processing) databases, organize data into a multidimensional structure called a cube. A cube is a data structure that has multiple dimensions, each representing a different aspect of the data, such as time, location, or product. Each dimension is made up of a hierarchy of members, such as days, months, and years for the time dimension, or countries, states, and cities for the location dimension.
- ◆ The process of building an MDDB summarizes the raw data; the data stored in the MDDB is thus said to be presummarized. The MDDB enables users to quickly retrieve multiple levels of presummarized data through a multidimensional view.

# Multidimensional Databases

| Time | Location="Kolkata" |     |       | Location="Delhi" |     |       | Location="Mumbai" |     |       |
|------|--------------------|-----|-------|------------------|-----|-------|-------------------|-----|-------|
|      | item               |     |       | item             |     |       | item              |     |       |
|      | Milk               | Egg | Bread | Milk             | Egg | Bread | Milk              | Egg | Bread |
| Q1   | 340                | 604 | 38    | 335              | 365 | 35    | 336               | 484 | 80    |
| Q2   | 680                | 583 | 10    | 684              | 490 | 48    | 595               | 594 | 39    |
| Q3   | 535                | 490 | 50    | 389              | 385 | 15    | 366               | 385 | 20    |

*3D data representation as 2D*



# Parallel Databases

- ◆ Nowadays organizations need to handle a huge amount of data with a high transfer rate. For such requirements, the client-server or centralized system is not efficient. With the need to improve the efficiency of the system, the concept of the parallel database comes in picture. A parallel database system seeks to improve the performance of the system through parallelizing concept.
- ◆ **A parallel database** is one which involves multiple processors and working in parallel on the database used to provide the services.
- ◆ A parallel database system seeks to improve performance through parallelization of various operations, such as loading data, building indexes and evaluating queries

# Parallel Databases

- ◆ Parallel DBMS is a Database Management System that runs through multiple processors and disks. They combine two or more processors also disk storage that helps make operations and executions easier and faster.
- ◆ **Parallel Databases** are designed to execute concurrent operations. They exist, happen, or done at the same time even if the data processed are not from one source or one processing unit.
- ◆ Although data may be stored in a distributed fashion, the distribution is governed solely by performance considerations. Parallel databases improve processing and input/output speeds by using multiple CPUs and disks in parallel.
- ◆ In parallel processing, many operations are performed simultaneously, as opposed to serial processing, in which the computational steps are performed sequentially.

# Parallel Databases

- **Intraquery parallelism**
  - A single query that is executed in parallel using multiple processors or disks.
- **Independent parallelism**
  - Execution of each operation individually in different processors only if they can be executed independent of each other. For example, if we need to join four tables, then two can be joined at one processor and the other two can be joined at another processor. Final join can be done later.

# Parallel Databases

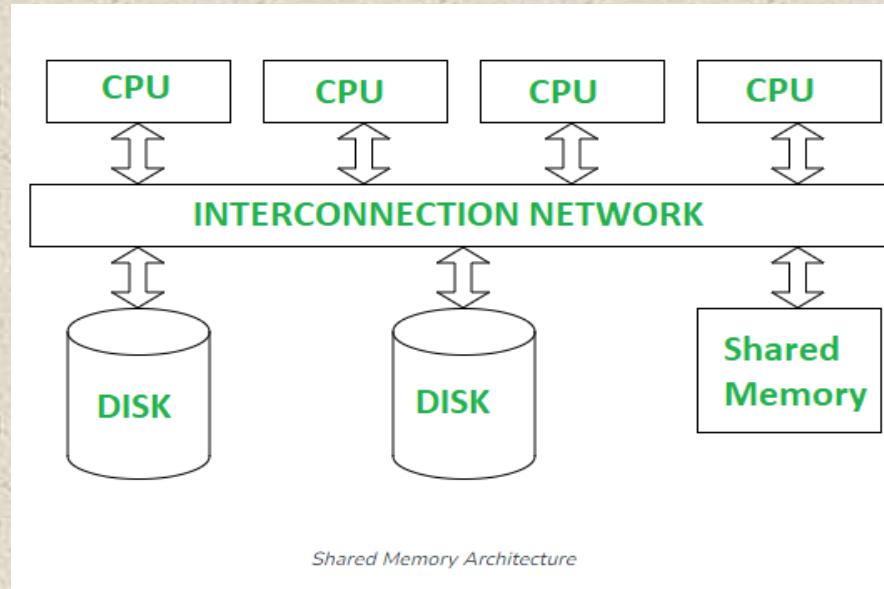
- **Pipe-lined parallelism**
  - Execution of different operations in pipe-lined fashion. For example, if we need to join three tables, one processor may join two tables and send the result set records as and when they are produced to the other processor. In the other processor the third table can be joined with the incoming records and the final result can be produced.
- **Intraoperation parallelism**
  - Execution of single complex or large operations in parallel in multiple processors. For example, ORDER BY clause of a query that tries to execute on millions of records can be parallelized on multiple processors.

# Parallel Databases

- ◆ Multiple resources like CPUs and Disks are used in parallel. The operations are performed simultaneously, as opposed to serial processing. A parallel server can allow access to a single database by users on multiple machines. It also performs many parallelization operations like data loading, query processing, building indexes, and evaluating queries. Parallel databases may have following architectures:
  - Shared Memory Architecture
  - Shared Disk Architecture
  - Shared Nothing Architecture
  - Hierarchical Architecture

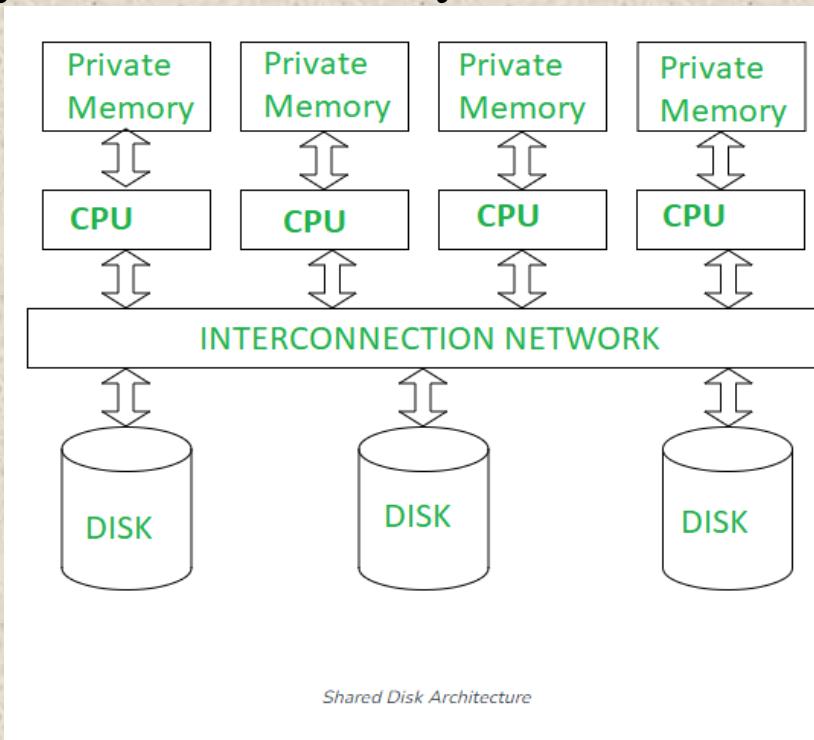
# Parallel Databases

- ◆ **Shared Memory Architecture-** In Shared Memory Architecture, there are multiple CPUs that are attached to an interconnection network. They are able to share a single or global main memory and common disk arrays. It is to be noted that, In this architecture, a single copy of a multi-threaded operating system and multithreaded DBMS can support these multiple CPUs. Also, the shared memory is a solid coupled architecture in which multiple CPUs share their memory. It is also known as Symmetric multiprocessing (SMP).



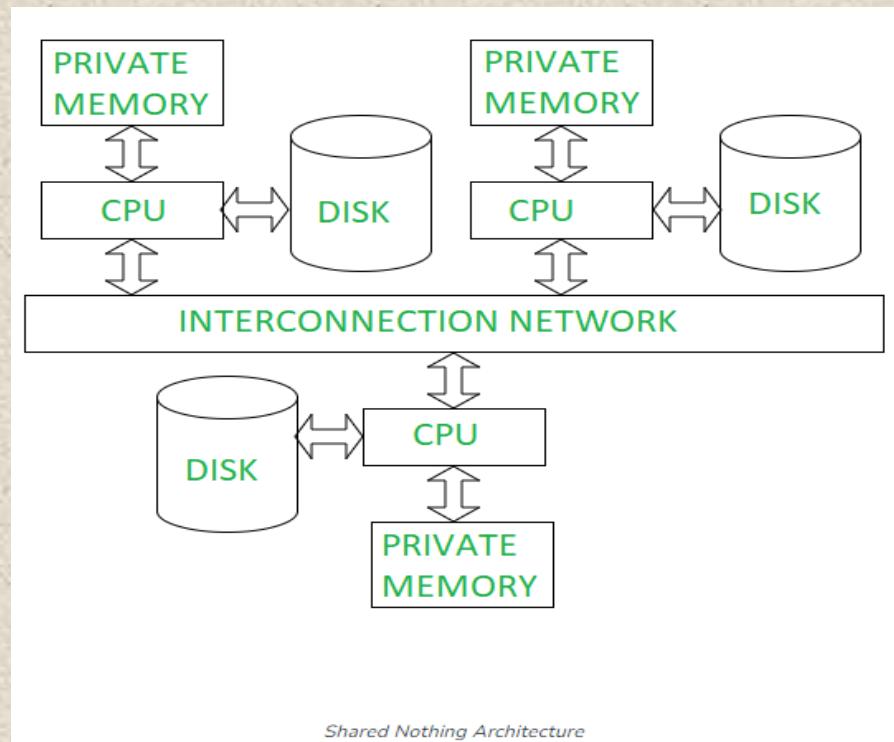
# Parallel Databases

- **Shared Disk Architectures :** In Shared Disk Architecture, various CPUs are attached to an interconnection network. In this, each CPU has its own memory and all of them have access to the same disk. Also, note that here the memory is not shared among CPUs therefore each node has its own copy of the operating system and DBMS. Shared disk architecture is a loosely coupled architecture optimized for applications that are inherently centralized. They are also known as **clusters**.



# Parallel Databases

- **Shared Nothing Architecture :** Shared Nothing Architecture is multiple processor architecture in which each processor has its own memory and disk storage. In this, multiple CPUs are attached to an interconnection network through a node. Also, note that no two CPUs can access the same disk area. In this architecture, no sharing of memory or disk resources is done. It is also known as Massively parallel processing (MPP).



# Parallel Databases

- **Hierarchical Architecture :** This architecture is a combination of shared disk, shared memory and shared nothing architectures. This architecture is scalable due to availability of more memory and many processor. But is costly to other architecture.

# Multimedia Databases

- ◆ Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as pictures and drawings), video clips (such as movies, newsreels, and home videos), audio clips (such as songs, phone messages, and speeches), and documents (such as books and articles).
- ◆ A Multimedia Database (MMDB) hosts one or more multimedia data types (i.e. text, images, graphic objects, audio, video, animation sequences). These data types are broadly categorized into three classes:
  - Static media (time-independent: image and graphic object).
  - Dynamic media (time-dependent: audio, video and animation).
  - Dimensional media(3D game and computer aided drafting programs).

# Multimedia Databases

- ◆ The contents of the multimedia database has additional information related to the primary multimedia data. To effectively manage and query a vast collection of multimedia data. These contents are –
- ◆ Media data
  - ◆ It is actual multimedia data or primary data stored in the multimedia database.
  - ◆ It represents a multimedia object. It can be an image, audio, video, animation, graphic object, or text.
- ◆ Media format data
  - ◆ It is information related to the format of the multimedia data such as frame rates and encoding schemes.
- ◆ Media keyword data
  - ◆ It is also known as content descriptive data. This information pertains to the generation of multimedia data, such as date and time in the case of images and videos.
- ◆ Media feature data
  - ◆ This data is used to describe the characteristics of multimedia data, such as the color distribution.

# Multimedia Databases

- ◆ The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest.
- ◆ For example, one may want to locate all video clips in a video database that include a certain person, say Michael Jackson. One may also want to retrieve video clips based on certain activities included in them, such as video clips where a soccer goal is scored by a certain player or team.
- ◆ The above types of queries are referred to as content-based retrieval, because the multimedia source is being retrieved based on its containing certain objects or activities.

# Multimedia Databases

- ◆ A multimedia database must use some model to organize and index the multimedia sources based on their contents. Identifying the contents of multimedia sources is a difficult and time-consuming task. There are two main approaches. The **content based** is based on automatic analysis of the multimedia sources to identify certain mathematical characteristics of their contents. This approach uses different techniques depending on the type of multimedia source (image, video, audio, or text).
- ◆ The second approach depends on **manual identification** of the objects and activities of interest in each multimedia source and on using this information to index the sources. This approach can be applied to all multimedia sources, but it requires a manual preprocessing phase in which a person must scan each multimedia source to identify and catalog the objects and activities it contains so that they can be used to index the sources.

# Multimedia Databases

- ◆ A typical **image database query** would be to find images in the database that are similar to a given image. The given image could be an isolated segment that contains, say, a pattern of interest, and the query is to locate other images that contain that same pattern.
- ◆ There are two main techniques for this type of search. The first approach uses a **distance function** to compare the given image with the stored images and their segments. If the distance value returned is small, the probability of a match is high. Indexes can be created to group stored images that are close in the distance metric so as to limit the search space.
- ◆ The second approach, called the **transformation approach**, measures image similarity by having a small number of transformations that can change one image's cells to match the other image. Transformations include rotations, translations, and scaling. Although the transformation approach is more general, it is also more time consuming and difficult.

# Multimedia Databases

- ◆ The multimedia database must support large objects, since multimedia data such as videos can occupy up to a few gigabytes of storage.
- ◆ Many database systems do not support objects larger than a few gigabytes.
- ◆ Larger objects could be split into smaller pieces and stored in the database.
- ◆ Alternatively, the multimedia object may be stored in a file system, but the database may contain a pointer to the object; the pointer would typically be a file name.
- ◆ The SQL/MED standard (Management of External Data) allows external data, such as files, to be treated as if they are part of the database. With SQL/MED, the object would appear to be part of the database, but can be stored externally.

# Temporal Databases

- ◆ Temporal databases permit the database system to store a history of changes and allow users to query both current and past states of the database. Some temporal database models also allow users to store future expected information, such as planned schedules.
- ◆ Databases that store information about states of the real world across time are called temporal databases.

# Temporal Databases

- A temporal relation is one where each tuple has an associated time when it is true; the time may be either valid time or transaction time.

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> | <i>from</i> | <i>to</i>  |
|-----------|-------------|------------------|---------------|-------------|------------|
| 10101     | Srinivasan  | Comp. Sci.       | 61000         | 2007/1/1    | 2007/12/31 |
| 10101     | Srinivasan  | Comp. Sci.       | 65000         | 2008/1/1    | 2008/12/31 |
| 12121     | Wu          | Finance          | 82000         | 2005/1/1    | 2006/12/31 |
| 12121     | Wu          | Finance          | 87000         | 2007/1/1    | 2007/12/31 |
| 12121     | Wu          | Finance          | 90000         | 2008/1/1    | 2008/12/31 |
| 98345     | Kim         | Elec. Eng.       | 80000         | 2005/1/1    | 2008/12/31 |

**Figure 25.1** A temporal *instructor* relation.

# Spatial Databases

- ◆ Spatial databases incorporate functionality that provides support for databases that keep track of objects in a multidimensional space.
- ◆ For example, cartographic databases that store maps include two-dimensional spatial descriptions of their objects— from countries and states to rivers, cities, roads, seas, and so on.
- ◆ The systems that manage geographic data and related applications are known as Geographic Information Systems (GISs).

# Spatial Databases

- ◆ In general, a spatial database stores objects that have spatial characteristics that describe them and that have spatial relationships among them.
- ◆ A spatial database is optimized to store and query data related to objects in space, including points, lines and polygons.
- ◆ Queries posed on these spatial data, where predicates for selection deal with spatial parameters, are called spatial queries. For example, “What are the names of all bookstores within five miles of the College of Computing building at Georgia Tech?” is a spatial query

# Analytical Operations on Spatial Databases

- ◆ Measurement operations are used to measure some global properties of single objects (such as the area, the relative size of an object's parts, compactness, or symmetry) and to measure the relative position of different objects in terms of distance and direction.
- ◆ **Spatial analysis operations**, which often use statistical techniques, are used to uncover spatial relationships within and among mapped data layers. An example would be to create a map—known as a prediction map—that identifies the locations of likely customers for particular products based on the historical sales and demographic information.
- ◆ **Flow analysis operations** help in determining the shortest path between two points and also the connectivity among nodes or regions in a graph.
- ◆ **Location analysis** aims to find if the given set of points and lines lie ~~within~~ within a given polygon (location).

# Mobile Databases

- ◆ A Mobile database is a database that can be connected to a mobile computing device over a mobile network (or wireless network). Here the client and the server have wireless connections. In today's world, mobile computing is growing very rapidly, and it is huge potential in the field of the database. It will be applicable on different-different devices like android based mobile databases, iOS based mobile databases, etc. Common examples of databases are *SQLite*, *Couch base Lite*, *Object Box*, etc.
- ◆ Mobile Database typically involves three parties :
  - ◆ **Fixed Hosts** – It performs the transactions and data management functions with the help of database servers.
  - ◆ **Mobile Units** – These are portable computers that move around a geographical region that includes the cellular network that these units use to communicate to base stations.
  - ◆ **Base Stations** – These are two-way radios installation in fixed locations, that pass communication with the mobile units to and from the fixed hosts.

# Mobile Databases

- **Mobile Database Issues**
  - Data Management
    - Data Caching
    - Data Broadcast
    - Data Classification (Location Dependent/ Independent)
  - Transaction Management
    - Query processing
    - Transaction processing
    - Concurrency control
    - Database recovery

# Web Databases

- A **Web database** is a database application designed to be managed and accessed through the Internet. Website operators can manage this collection of data and present analytical results based on the data in the Web database application.
- A web database is a system for storing and displaying information that is accessible from the Internet / web. The database might be used for any of a wide range of functions, such as a membership database, client list, or inventory database.
- Content management systems commonly use web databases to store information such as posts, usernames, and comments. Using a database allows the website to be updated easily and without the need to edit the HTML code for each individual page. Not only is this a much more efficient way of creating and updating a website, but it also makes the process more accessible to people who aren't fluent in the programming languages of the Internet.

# Web Databases

- A web database is ideal for situations when the information should be shared, or when it must be accessed from various locations. It is especially beneficial when the system is to be shared between locations or different devices.
- Businesses both large and small can use Web databases to create website polls, feedback forms, client or customer and inventory lists. Personal Web database use can range from storing personal email accounts to a home inventory to personal website analytics. The Web database is entirely customizable to an individual's or business's needs.

# **Data Warehouse, Data Mining and Data Mart**

- ◆ Will Cover in Next Unit 7.

# Unit - 7

# Advanced Topics

# Concept of Object-Oriented Database

- ◆ Extend the relational data model by including object orientation and constructs to deal with added data types.
- ◆ Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- ◆ Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- ◆ Permit non-atomic domains (atomic  $\equiv$  indivisible)
- ◆ Example of non-atomic domain: set of integers, or set of tuples
- ◆ Allows more intuitive modeling for applications with complex data

# Concept of Object-Oriented Database

- ◆ Loosely speaking, an object corresponds to an entity in the ER model.
- ◆ The object-oriented paradigm is based on encapsulating code/method and data related to an object into single unit.
- ◆ The object-oriented data model is a logical data model (like the E-R model)

# Concept of Object-Oriented Database

- ◆ A core object-oriented data model consists of the following basic object-oriented concepts:
  - Object and object identifier: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).
  - Attributes and methods: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.
  - Class: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.

# Concept of Object-Oriented Database

- ◆ A core object-oriented data model consists of the following basic object-oriented concepts:
  - Class hierarchy and inheritance: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

| Object 1: Maintenance Report |  | Object 1 Instance |
|------------------------------|--|-------------------|
| Date                         |  | 01-12-01          |
| Activity Code                |  | 24                |
| Route No.                    |  | I-95              |
| Daily Production             |  | 2.5               |
| Equipment Hours              |  | 6.0               |
| Labor Hours                  |  | 6.0               |

| Object 2: Maintenance Activity |  |
|--------------------------------|--|
| Activity Code                  |  |
| Activity Name                  |  |
| Production Unit                |  |
| Average Daily Production Rate  |  |

# Concept of Object-Oriented Database

- ◆ An object has associated with it:
  - A set of variables that contain the data for the object. The value of each variable is itself an object.
  - A set of messages to which the object responds; each message may have zero, one, or more parameters.
  - A set of methods, each of which is a body of code to implement a message; a method returns a value as the response to the message

```
CREATE TABLE cities (
 name text,
 population float,
 altitude int -- in feet
);
```

```
CREATE TABLE capitals (
 state char(2)
) INHERITS (cities);
```

# Database Security

- ◆ Database security is a broad area that addresses many issues, including the following:
  - **Various legal and ethical issues** regarding the right to access certain information—for example, some information may be deemed to be private and cannot be accessed legally by unauthorized organizations or persons.
  - **Policy issues** at the governmental, institutional, or corporate level regarding what kinds of information should not be made publicly available—for example, credit ratings and personal medical records.
  - **System-related issues** such as the system levels at which various security functions should be enforced—for example, whether a security function should be handled at the physical hardware level, the operating system level, or the DBMS level.
- ◆ The need in some organizations to identify multiple security levels and to categorize the data and users based on these classifications—for example, top secret, secret, confidential, and unclassified. The security policy of the organization with respect to permitting access to various classifications of data must be enforced.

# Threats to Database

- ◆ Threats to databases can result in the loss or degradation of some or all of the following commonly accepted security goals: integrity, availability, and confidentiality.
  - **Loss of integrity:** Database integrity refers to the requirement that information be protected from improper modification. Modification of data includes creating, inserting, and updating data; changing the status of data; and deleting data. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts. If the loss of system or data integrity is not corrected, continued use of the contaminated system or corrupted data could result in inaccuracy, fraud, or erroneous decisions.
  - **Loss of availability:** Database availability refers to making objects available to a human user or a program who/which has a legitimate right to those data objects. Loss of availability occurs when the user or program cannot access these objects.
  - **Loss of confidentiality:** Database confidentiality refers to the protection of data from unauthorized disclosure. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization.

# Database Access Control

- ◆ A DBMS typically includes a database security and authorization subsystem that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:
  - **Discretionary security mechanisms:** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
  - **Mandatory security mechanisms:** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization. For example, a typical security policy is to permit users at a certain classification (or clearance) level to see only the data items classified at the user's own (or lower) classification level. An extension of this is role-based security, which enforces policies and privileges based on the concept of organizational roles

# Database Security

- ◆ It is now customary to refer to two types of database security mechanisms:
- ◆ **Discretionary security mechanisms:** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
- ◆ **Mandatory security mechanisms:** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization.

# Control Measures

- ◆ Four main control measures are used to provide security of data in databases:
  - Access control
  - Inference control
  - Flow control
  - Data encryption
- ◆ Above control measures define the level of access protections.

# Control Measures

- ◆ **Access Control** is a way of preventing unauthorized persons from accessing the system itself, either to obtain information or to make malicious changes in a portion of the database. The security mechanism of a DBMS must include provisions for restricting access to the database system as a whole.
- ◆ **Statistical databases** are used to provide statistical information or summaries of values based on various criteria. Security for statistical databases must ensure that information about individuals cannot be accessed. It is sometimes possible to deduce or infer certain facts concerning individuals from queries that involve only summary statistics on groups; consequently, this must not be permitted either. The control measure used to prevent is known as **inference control**.
- ◆ Another security issue is that of **flow control, which prevents information from** flowing in such a way that it reaches unauthorized users.

# Control Measures

- ◆ A final control measure is **data encryption**, which is used to protect sensitive data (such as credit card numbers) that is transmitted via some type of communications network. Encryption can be used to provide additional protection for sensitive portions of a database as well. The data is **encoded using some coding algorithm, known as encryption algorithm**. An unauthorized user who accesses encoded data will have difficulty decoding/deciphering it, but authorized users are given decoding or decrypting algorithms to decipher the data. Both of the encoding and decoding algorithms use some parameters like key for encrypting and decrypting data.

# User Accounts and Database Audits

- ◆ Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account number and password for the user if there is a legitimate need to access the database**. The user must **log in to the DBMS by entering the account number and password** whenever database access is needed. The DBMS checks that the account number and password are valid; if they are, the user is permitted to use the DBMS and to access the database.
- ◆ It is straightforward to keep track of database users and their accounts and passwords by creating an encrypted table or file with two fields: Account Number and Password. This table can easily be maintained by the DBMS. Whenever a new account is created, a new record is inserted into the table. When an account is canceled, the corresponding record must be deleted from the table.

# User Accounts and Database Audits

- ◆ The database system must also keep track of all operations on the database that are applied by a certain user throughout each **login session**, which *consists of the sequence of database interactions that a user performs from the time of logging in to the time of logging off.*
- ◆ To keep a record of all updates applied to the database and of particular users who applied each update, the **system log is maintained**. The **system log includes an entry for each operation applied to the database** that may be required for recovery from a transaction failure or system crash.
- ◆ If any tampering with the database is suspected, a **database audit is performed**, which consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period.

# User Accounts and Database Audits

- ◆ When an illegal or unauthorized operation is found, the DBA can determine the account number used to perform the operation. Database audits are particularly important for sensitive databases that are updated by many transactions and users, such as a banking database that can be updated by thousands of bank tellers.
- ◆ A database log that is used mainly for security purposes serves as an **audit trail**.

# Discretionary Access Control

- ◆ Discretionary access control is based on the concept of access rights (also called privileges) and mechanism for giving users such privileges. It grants the privileges (access rights) to users on different objects, including the capability to access specific data files, records or fields in a specified mode, such as, read, insert, delete or update or combination of these. A user who creates a database object such as a table or a view automatically gets all applicable privilege on that object. The DBMS keeps track of how these privileges are granted to other users.
- ◆ The typical method of enforcing **discretionary access control** in a database system is based on the **granting** and **revoking** privileges.

# Discretionary Access Control

- ◆ Discretionary Access Control (DAC) enforces security by means of user identifiers(uid) and group identifiers (gid); only the owner of the data (i.e., the Content Provider) holds the r/w permissions on the file.
- ◆ Each data object on a DAC based system has an *Access Control List* (ACL) associated with it. An ACL contains a list of users and groups to which the user has permitted access together with the level of access for each user or group. For example, *User A* may provide read-only access on one of her files to *User B*, read and write access on the same file to *User C* and full control to any user belonging to *Group 1*.

# Discretionary Access Control

- ◆ The **account level**:
  - At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- ◆ The **relation level** (or **table level**):
  - At this level, the DBA can control the privilege to access each individual relation or view in the database.

# Discretionary Access Control

- ◆ The privileges at the **account level** apply to the capabilities provided to the account itself and can include
  - the **CREATE SCHEMA** or **CREATE TABLE** privilege, to create a schema or base relation;
  - the **CREATE VIEW** privilege;
  - the **ALTER** privilege, to apply schema changes such adding or removing attributes from relations;
  - the **DROP** privilege, to delete relations or views;
  - the **MODIFY** privilege, to insert, delete, or update tuples;
  - and the **SELECT** privilege, to retrieve information from the database by using a **SELECT** query.

# Discretionary Access Control

- ◆ The privileges at the **relation level** apply base relations and views.
  - Each relation R in a database is assigned an **owner account**, which is typically the account that was used when the relation was created in the first place.
  - The owner of a relation is given all privileges on that relation.
  - The owner account holder can **pass privileges** on any of the owned relation to other users by **granting** privileges to their accounts.

# Discretionary Access Control

- ◆ In **Relation Level**, following types of privileges can be granted on each individual relation  $R$ :
  - **SELECT (retrieval or read) privilege on  $R$ :** *Gives the account retrieval privilege.* In SQL, this gives the account the privilege to use the SELECT statement to retrieve tuples from  $R$ .
  - **Modification privileges on  $R$ :** *This gives the account the capability to modify the tuples of  $R$ . In SQL, this includes three privileges: UPDATE, DELETE, and INSERT.*
  - **References privilege on  $R$ :** *This gives the account the capability to reference (or refer to) a relation  $R$  when specifying integrity constraints. This privilege can also be restricted to specific attributes of  $R$ .*

# Discretionary Access Control

- ◆ **Granting of Privileges:**
- ◆ Whenever the owner A of a relation R grants a privilege on R to another account B, the privilege can be given to B with or without the **GRANT OPTION**. If the **GRANT OPTION** is given, this means that B can also grant that privilege on R to other accounts otherwise not.

# Discretionary Access Control

- ◆ **Revoking of Privileges:**
- ◆ In some cases, it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking privileges is needed.** In SQL, a **REVOKE** command is included for the purpose of canceling privileges.

# Discretionary Access Control

## ◆ Examples of Granting and Revoking

- GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;
- GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH GRANT OPTION; (A3 can propagate grants to others)
- GRANT SELECT ON EMPLOYEE TO A4; (A4 can't propagate grant to others)
- REVOKE SELECT ON EMPLOYEE FROM A3;

# Mandatory Access Control

- ◆ Mandatory Access Control (MAC) is based on clearance, i.e., security labels (secret, top secret, confidential, etc.). Data objects are given a security classification, and the user will be denied access if his clearance is lower than the classification of the object.
- ◆ Similarly, each user account on the system also has classification and category properties from the same set of properties applied to the data objects. When a user attempts to access a data under Mandatory Access Control the system checks the user's classification and categories and compares them to the properties of the object's security label. If the user's credentials match the MAC security label properties of the data object access is allowed. It is important to note that *both* the classification and categories must match. A user with top secret classification, for example, cannot access a resource if they are not also a member of one of the required categories for that object.

# Mandatory Access Control

- ◆ Typical security classes
  - **Top secret (TS),**
  - **Secret (S),**
  - **Confidential (C),**
  - **Unclassified (U),**
- ◆ Here TS is the highest level and U the lowest:
- ◆ **TS > S > C > U**

# Mandatory Access Control

- ◆ **Subjects**

E.g., user, account, program

- ◆ **Objects**

E.g., Relation, tuple, column, view, operation.

- ◆ Subjects and Objects classified into, T, S, C, or U:
- ◆ **Clearance** (classification) of a subject S denoted as **class(S)** and to the **classification** of an object O as **class(O)**.

# Mandatory Access Control

- ◆ Two restrictions are enforced on data access based on the subject/object classifications:
- ◆ **Simple security property:** A subject  $S$  is not allowed read access to an object  $O$  unless  $\text{class}(S) \geq \text{class}(O)$ .
- ◆ **Star property:** A subject  $S$  is not allowed to write an object  $O$  unless  $\text{class}(S) \leq \text{class}(O)$ .
- ◆ The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance.
- ◆ The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance.

# Role Based Access Control

- ◆ Its basic notion is that privileges and other permissions are associated with organizational roles rather than with individual users. Individual users are then assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands.
- ◆ For example, a company may have roles such as sales account manager, purchasing agent, mailroom clerk, customer service manager, and so on. Multiple individuals can be assigned to each role. Security privileges that are common to a role are granted to the role name, and any individual assigned to this role would automatically have those privileges granted.

# Data Encryption and Decryption

- ◆ **Encryption** is the conversion of data into a form, called a **ciphertext**, that cannot be easily understood by **unauthorized persons**. It enhances security and privacy when access controls are bypassed, because in cases of data loss or theft, encrypted data cannot be easily understood by unauthorized persons.
- ◆ **Decryption** is the conversion Ciphertext back into plaintext that can be easily understood by authorized persons. It is a reverse process of encryption.
- ◆ **Cryptography** is an art of hiding text and includes **encryption and decryption process**.

# Data Encryption and Decryption

- ◆ **Ciphertext:** Encrypted (*enciphered*) data
- ◆ **Plaintext (or cleartext):** Intelligible data that has meaning and can be read or acted upon without the application of decryption
- ◆ **Encryption:** The process of transforming plaintext into ciphertext
- ◆ **Decryption:** The process of transforming ciphertext back into plaintext
- ◆ Encryption consists of applying an **encryption algorithm** to **data using some prespecified encryption key**. The resulting data must be decrypted using a **decryption key** to recover the original data.

# Data Encryption and Decryption

- ◆ **Private (Symmetric) Key Algorithms**
- ◆ A symmetric key is one key that is used for both encryption and decryption.
- ◆ By using a symmetric key, fast encryption and decryption is possible for routine use with sensitive data in the database.
- ◆ A message encrypted with a secret key can be decrypted only with the same secret key.
- ◆ Algorithms used for symmetric key encryption are called **secret key algorithms**. Since **secret-key algorithms** are mostly used for encrypting the content of a message, they are also called **content-encryption algorithms**.

# Data Encryption and Decryption

- ◆ **Private (Symmetric) Key Algorithms**
- ◆ The major liability associated with secret-key algorithms is the need for sharing the secret key. A possible method is to derive the secret key from a user-supplied password string by applying the same function to the string at both the sender and receiver; this is known as a *password-based encryption algorithm*.
- ◆ *The strength of the symmetric key* encryption depends on the size of the key used. For the same algorithm, encrypting using a longer key is tougher to break than the one using a shorter key.
- ◆ Eg: DES, AES etc.

# Data Encryption and Decryption

- ◆ **Public (Assymmetric) Key Algorithms**
- ◆ These algorithms **use two related different keys, a public key and a private key**, to perform encryption and decryption.
- ◆ The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication.
- ◆ The two keys used for public key encryption are referred to as the **public key and the private key**. The private key is kept secret, but it is referred to as a *private key rather than a secret key* (*the key* used in conventional encryption) to avoid confusion with conventional encryption. The two keys are mathematically related, since one of the keys is used to perform encryption and the other to perform decryption. However, it is very difficult to derive the private key from the public key.

# Data Encryption and Decryption

- ◆ **Public (Assymmetric) Key Algorithms**
- ◆ As the name suggests, **the public key of the pair is made public for others to use, whereas the private key is known only to its owner.**
- ◆ A general-purpose public key cryptographic algorithm relies on one key for encryption and a different but related key for decryption. The essential steps are as follows:
  - Each user generates a pair of keys to be used for the encryption and decryption of messages.
  - Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private.
  - If a sender wishes to send a private message to a receiver, the sender encrypts the message using the receiver's public key.
  - When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.
- ◆ Eg: RSA, ElGamal etc.

# Data Encryption and Decryption

- ◆ **Public (Assymmetric) Key Algorithms**
- ◆ A public key encryption scheme, or *infrastructure*, *has following ingredients:*
  - **Plaintext:** This is the data or readable message that is fed into the algorithm as input.
  - **Encryption algorithm:** This algorithm performs various transformations on the plaintext.
  - **Public and private keys:** These are a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input. For example, if a message is encrypted using the public key, it can only be decrypted using the private key.
  - **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
  - **Decryption algorithm:** This algorithm accepts the ciphertext and the matching key and produces the original plaintext

# Statistical Databases

- ◆ Statistical databases are used mainly to produce statistics about various populations. The database may contain confidential data about individuals; this information should be protected from user access. However, users are permitted to retrieve statistical information about the populations, such as averages, sums, counts, maximums, minimums, and standard deviations.
- ◆ **A population** is a set of tuples of a relation (table) that satisfy some selection condition.
- ◆ **Statistical queries** involve applying statistical functions to a population of tuples. For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person

# Statistical Databases

- ◆ Statistical database security techniques must prohibit the retrieval of individual data. This can be achieved by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called statistical queries.

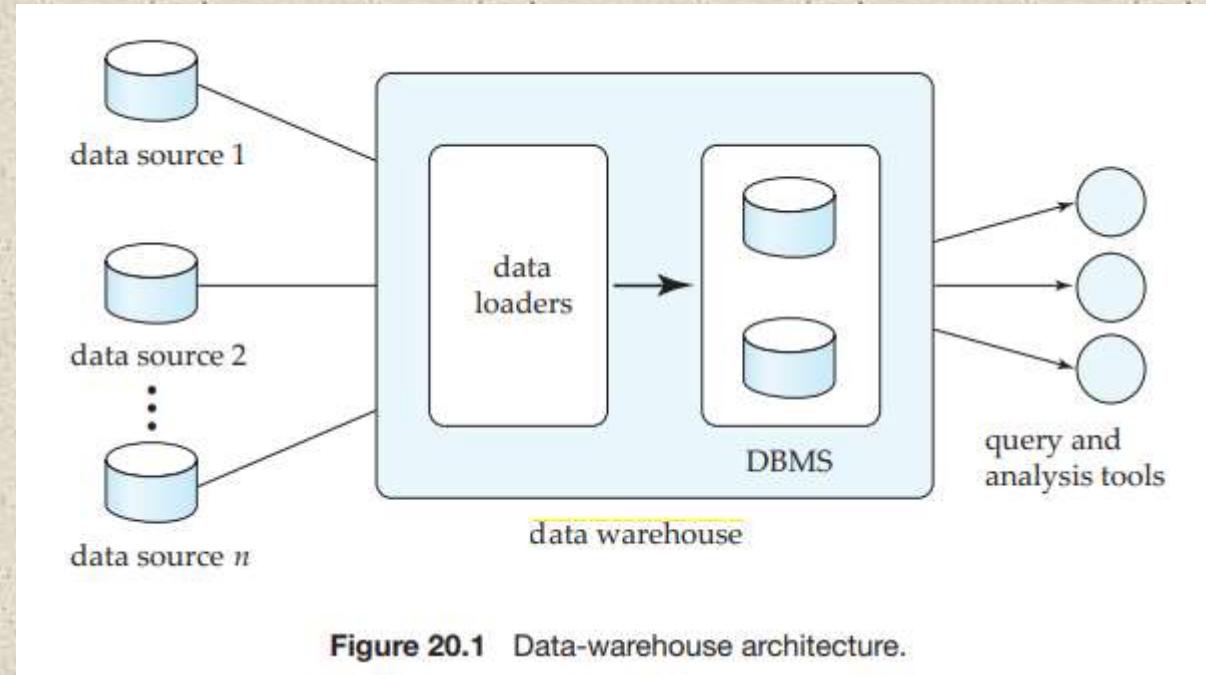
# Data Warehousing

- ◆ Data warehouses are databases that store and maintain analytical data separately from transaction-oriented databases for the purpose of decision support. Regular transaction oriented databases store data for a limited period of time before the data loses its immediate usefulness and it is archived.
- ◆ On the other hand, data warehouses tend to keep years' worth of data in order to enable analysis of historical data. They provide storage, functionality, and responsiveness to queries beyond the capabilities of transaction-oriented databases.
- ◆ A data warehouse is a repository of data gathered from multiple sources and stored under a common, unified database schema. Data stored in warehouse are analyzed by a variety of complex aggregations and statistical analyses.

# Data Warehousing

- ◆ A data warehouse is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write. Moreover, by accessing information for decision support from a data warehouse, the decision maker ensures that online transaction-processing systems are not affected by the decision-support workload.

# Data Warehousing



# Data Warehousing

- ◆ The different steps involved in getting data into a data warehouse are called extract, transform, and load or ETL tasks; extraction refers to getting data from the sources, while load refers to loading the data into the data warehouse.
- ◆ Data warehousing is the process of constructing and using a data warehouse. A data warehouse is constructed by integrating data from multiple heterogeneous sources that support analytical reporting, structured and/or ad hoc queries, and decision making.

# Data Warehousing

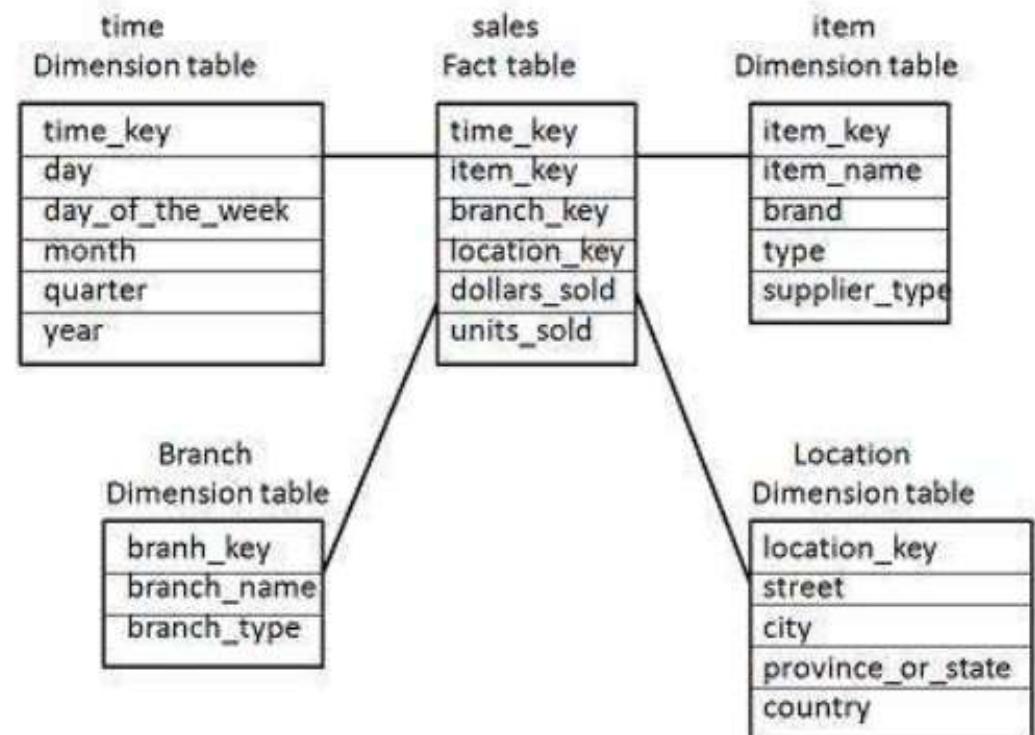
- ◆ Data warehouses are
  - **Subject-oriented:** They can analyze data about a particular subject or functional area (such as sales).
  - **Integrated:** Data warehouses create consistency among different data types from disparate sources.
  - **Nonvolatile:** Once data is in a data warehouse, it's stable and doesn't change.
  - **Time-variant:** Data warehouse analysis looks at change over time.

# Data Warehousing

- ◆ Schema is a logical description of the entire database. It includes the name and description of records of all record types including all associated data-items and aggregates.
- ◆ Much like a database, a data warehouse also requires to maintain a schema. A database uses relational model, while a data warehouse uses **Star, Snowflake, and Fact Constellation schema**.

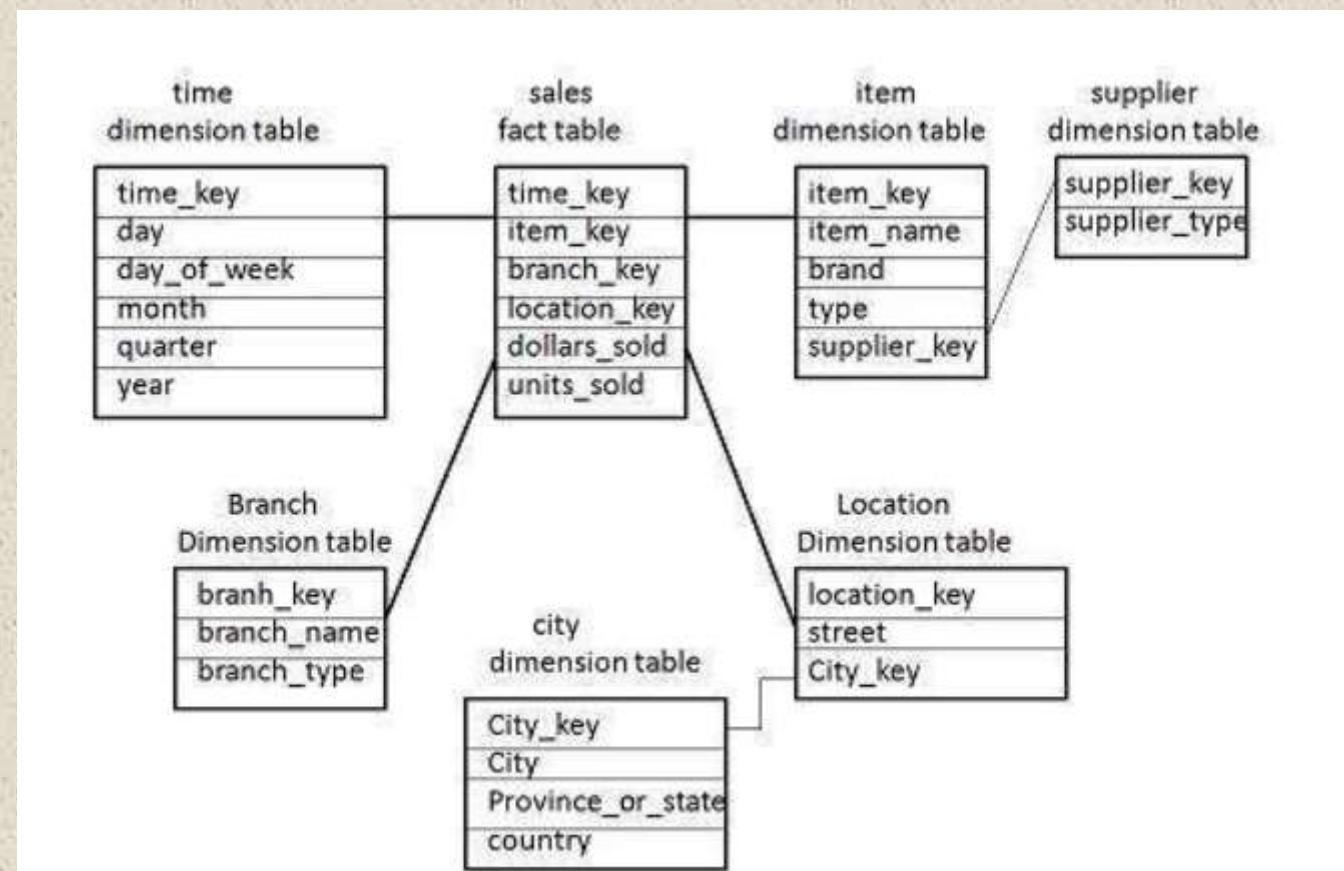
# Data Warehousing

- ◆ It has a fact table and number of dimension tables.
- ◆ The dimension tables contain the set of attributes.
- ◆ The fact table contains the keys to each of dimension tables. The fact table also contains the normal attributes.



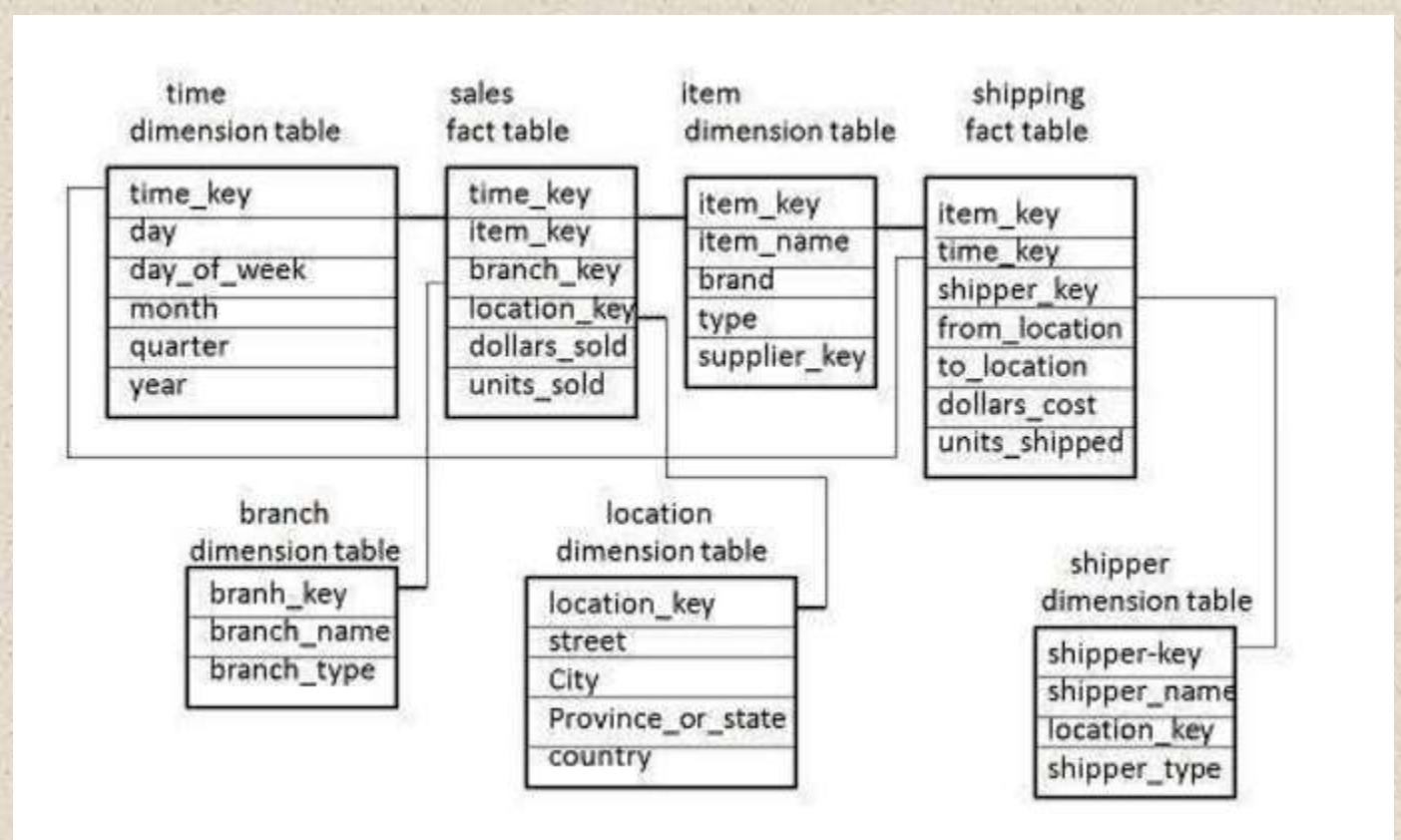
# Data Warehousing

- ◆ Snowflake Schema
- ◆ The snowflake schema consists of one fact table which is linked to many dimension tables, which can be linked to other dimension tables through a many-to-one relationship.



# Data Warehousing

- ◆ Fact Constellation Schema
- ◆ A fact constellation has multiple fact tables. It is also known as galaxy schema.



# Data Mining

- ◆ Data mining refers to the mining or discovery of new information in terms of patterns or rules from vast amounts of data. To be practically useful, data mining must be carried out efficiently on large files and databases.
- ◆ The goal of a data warehouse is to support decision making with data. Data mining can be used in conjunction with a data warehouse to help with certain types of decisions.
- ◆ Data mining helps in extracting meaningful new patterns that cannot necessarily be found by merely querying or processing data or meta-data in the data warehouse.

# Data Mining

- ◆ The result of mining may be to discover the following types of new information:
- ◆ **Association rules**—for example, whenever a customer buys video equipment, he or she also buys another electronic gadget.
- ◆ **Sequential patterns**—for example, suppose a customer buys a camera, and within three months he or she buys photographic supplies, then within six months he is likely to buy an accessory item. This defines a sequential pattern of transactions. A customer who buys more than twice in lean periods may be likely to buy at least once during the December holiday shopping period.
- ◆ **Classification trees**—for example, customers may be classified by frequency of visits, types of financing used, amount of purchase, or affinity for types of items; some revealing statistics may be generated for such classes.

# Data Mining

- ◆ **The classification problem:** Given that items belong to one of several classes, and given past instances (called training instances) of items along with the classes to which they belong, the problem is to predict the class to which a new item belongs. The class of the new instance is not known, so other attributes of the instance must be used to predict the class.
- ◆ **Classification** is the process of learning a model that describes different classes of data. The classes are predetermined. For example, in a banking application, customers who apply for a credit card may be classified as a poor risk, fair risk, or good risk. Hence this type of activity is also called **supervised learning**.

# Data Mining

- ◆ **Association information** can be used in several ways. When a customer buys a particular book, an online shop may suggest associated books. A grocery shop may decide to place bread close to milk, since they are often bought together, to help shoppers finish their task faster.
- ◆ An example of an association rule is: bread  $\Rightarrow$  milk.
- ◆ In the context of grocery-store purchases, the rule says that customers who buy bread also tend to buy milk with a high probability.
- ◆ This is known as **association rule mining**.
- ◆ A common example is that of market-basket data. Here the market basket corresponds to the sets of items a consumer buys in a supermarket during one visit.

# Data Mining

- ◆ **Clustering** refers to the problem of finding clusters of points in the given data.
- ◆ The problem of clustering can be formalized from distance metrics in several ways.
  - One way is to phrase it as the problem of grouping points into  $k$  sets (for a given  $k$ ) so that the average distance of points from the centroid of their assigned cluster is minimized.
  - Another way is to group points so that the average distance between every pair of points in each cluster is minimized.
- ◆ A given population of events or items can be partitioned (segmented) into sets of “similar” elements. For example: An entire population of treatment data on a disease may be divided into groups based on the similarity of side effects produced.

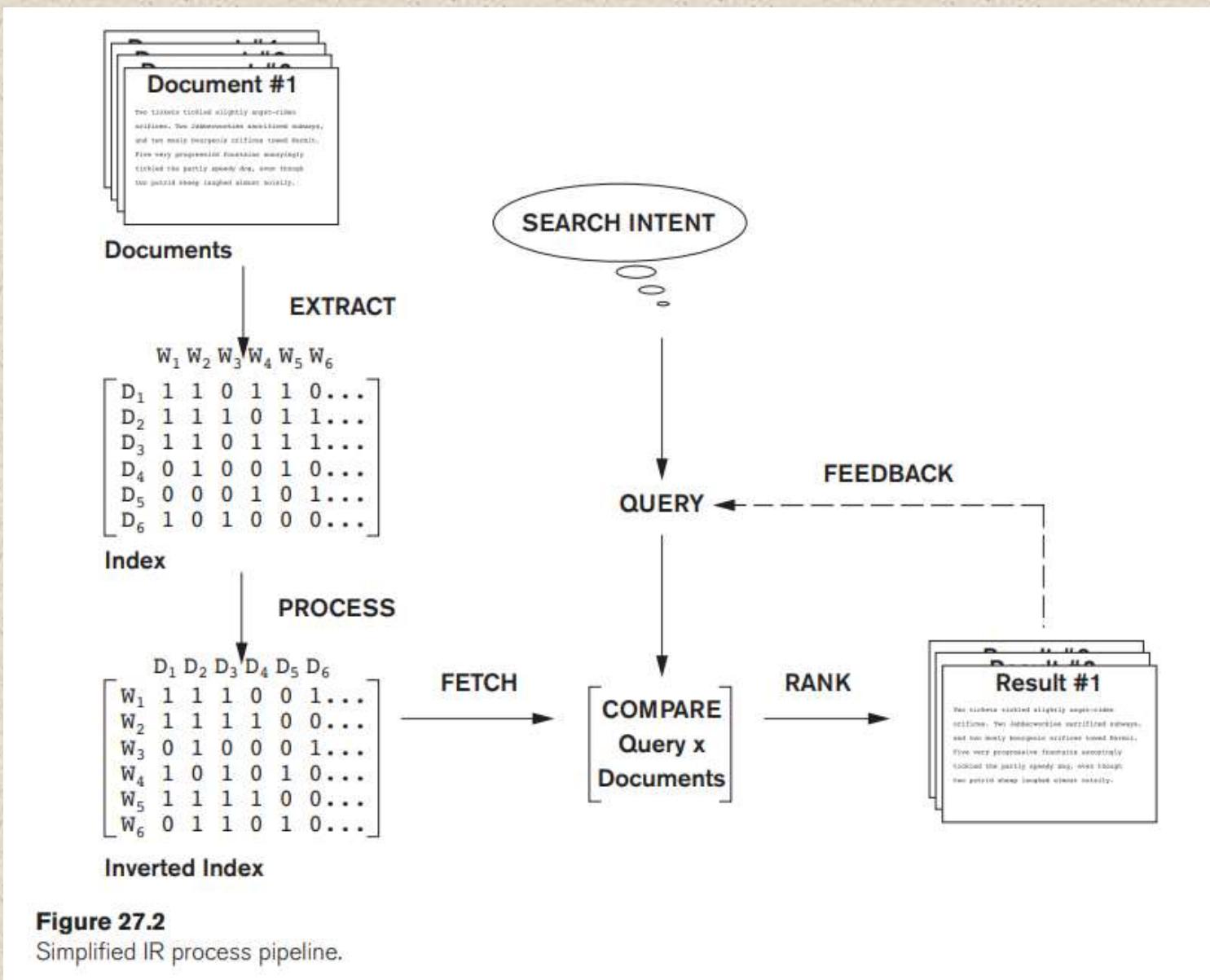
# Data Mining

- ◆ The previous data mining task of classification deals with partitioning data based on using a preclassified training sample. However, **Clustering** is often useful to partition data without having a training sample; this is also known as unsupervised learning.
- ◆ For example, in business, it may be important to determine groups of customers who have similar buying patterns, or in medicine, it may be important to determine groups of patients who show similar reactions to prescribed drugs.
- ◆ The goal of **clustering** is to place records into groups, such that records in a group are similar to each other and dissimilar to records in other groups. The groups are usually disjoint.

# Information Retrieval

- ◆ With the advent of the World Wide Web (or Web, for short), the volume of unstructured information stored in messages and documents that contain textual and multimedia information has exploded. These documents are stored in a variety of standard formats, including HTML, XML, and several audio and video formatting standards.
- ◆ **Information retrieval (IR)** deals with the problems of storing, indexing, and retrieving (searching) such information to satisfy the needs of users. The problems that IR deals with are exacerbated by the fact that the number of Web pages and the number of social interaction events is already in the billions and is growing at a phenomenal rate.
- ◆ Information retrieval deals mainly with unstructured data, and the techniques for indexing, searching, and retrieving information from large collections of unstructured documents.

# Information Retrieval



**Figure 27.2**  
Simplified IR process pipeline.

# Information Retrieval

- ◆ With the advent of the World Wide Web (or Web, for short), the volume of unstructured information stored in messages and documents that contain textual and multimedia information has exploded. These documents are stored in a variety of standard formats, including HTML, XML, and several audio and video formatting standards.
- ◆ **Information retrieval (IR)** deals with the problems of storing, indexing, and retrieving (searching) such information to satisfy the needs of users. The problems that IR deals with are exacerbated by the fact that the number of Web pages and the number of social interaction events is already in the billions and is growing at a phenomenal rate.
- ◆ Information retrieval deals mainly with unstructured data, and the techniques for indexing, searching, and retrieving information from large collections of unstructured documents.

# Relevance Ranking

- ◆ Using Terms
  - TF-IDF
  - Similarity Based
- ◆ Using Hyperlinks
  - Popularity Ranking
  - PageRank
- ◆ Using Synonyms, Homonyms, and Ontologies

# Crawling and Indexing the Web

- ◆ **Web crawlers** are programs that locate and gather information on the Web. They recursively follow hyperlinks present in known documents to find other documents.
- ◆ Crawlers start from an initial set of URLs, which may be created manually. Each of the pages identified by these URLs are fetched from the Web. The Web crawler then locates all URL links in these pages, and adds them to the set of URLs to be crawled, if they have not already been fetched, or added to the to-be-crawled set.
- ◆ This process is again repeated by fetching all pages in the to-be-crawled set, and processing the links in these pages in the same fashion. By repeating the process, all pages that are reachable by any sequence of links from the initial set of URLs would be eventually fetched.

# Crawling and Indexing the Web

- ◆ Since the number of documents on the Web is very large, it is not possible to crawl the whole Web in a short period of time; and in fact, all search engines cover only some portions of the Web, not all of it, and their crawlers may take weeks or months to perform a single crawl of all the pages they cover.
- ◆ There are usually many processes, running on multiple machines, involved in crawling. A database stores a set of links (or sites) to be crawled; it assigns links from this set to each crawler process.
- ◆ New links found during a crawl are added to the database, and may be crawled later if they are not crawled immediately. Pages have to be refetched (that is, links recrawled) periodically to obtain updated information, and to discard sites that no longer exist, so that the information in the **search index** is kept reasonably up-to-date.

# Crawling and Indexing the Web

- ◆ Pages fetched during a crawl are handed over to a prestige computation and indexing system, which may be running on a different machine. The prestige computation and indexing systems themselves run on multiple machines in parallel.
- ◆ Pages can be discarded after they are used for prestige computation and added to the index; however, they are usually cached by the search engine, to give search engine users fast access to a cached copy of a page, even if the original Web site containing the page is not accessible.
- ◆ To support very high query rates, the indices may be kept in main memory, and there are multiple machines; the system selectively routes queries to the machines to balance the load among them. Popular search engines often have tens of thousands of machines carrying out the various tasks of crawling, indexing, and answering user queries

# XML Databases

- ◆ Store data in **XML(Extensible Markup Language) format.**
- ◆ XML is a meta-markup language used to manage data which employs user customizable tags to organize information. The flexibility of the language, which allows the creation of custom data structures and organizational systems, has led to its widespread use to exchange data in multiple forms.
- ◆ An XML database uses a special programming language designed specifically to extract and manipulate XML documents, known as Xquery together with XPath. The purpose of XQuery is to allow the construction of flexible queries that can extract and manipulate information from XML documents, as well as other sources that can be translated into XML. XPath is used to navigate through elements and attributes in XML.

# XML Databases

- ◆ There are two major categories of these databases: **XML-enabled databases** and **Native XML databases (NXD)**. Each type of XML database is used to store different types of data.
- ◆ An **XML-enabled database** funnels data into a traditional relational database in an XML format. The data is translated for storage, and returned to its initial format upon output. This type of database is used to store data-centric documents which include highly structured information, such as patient records, and only use XML for data transfer.
- ◆ **Native XML databases** store XML documents as a whole, instead of separating out the data within them, and are designed to store semi-structured information. Native XML databases have an advantage over the XML-enabled database, as it is easier to store, query and maintain the XML document in a native database than in a XML-enabled database. Instead of table format, Native XML database is based on container format.

# XML Databases

Sample XML data note.xml.

```
<remainder>
 <note>
 <to>Tove</to>
 <from>Jeni</from>
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
 </note>
 <note>
```

Xquery to display content in body of the XML element sent by Jeni:

```
for $x in doc("note.xml")/remainder/note
 where $x/from="Jeni"
 return $x/body
```