



A T M E
College of Engineering



DATABASE MANAGEMENT SYSTEMS (21CS53)

MODULE 1



Databases and Database Management Systems



Topics

- | **Basic Definitions**
- | **Example of a Database**
- | **Main Characteristics of Database Technology**
- | **Additional Benefits of Database Technology**
- | **When Not to Use a DBMS**
- | **Data Models**
 - **History of data Models**
 - **Network Data Model**
 - **Hierarchical Data Model**

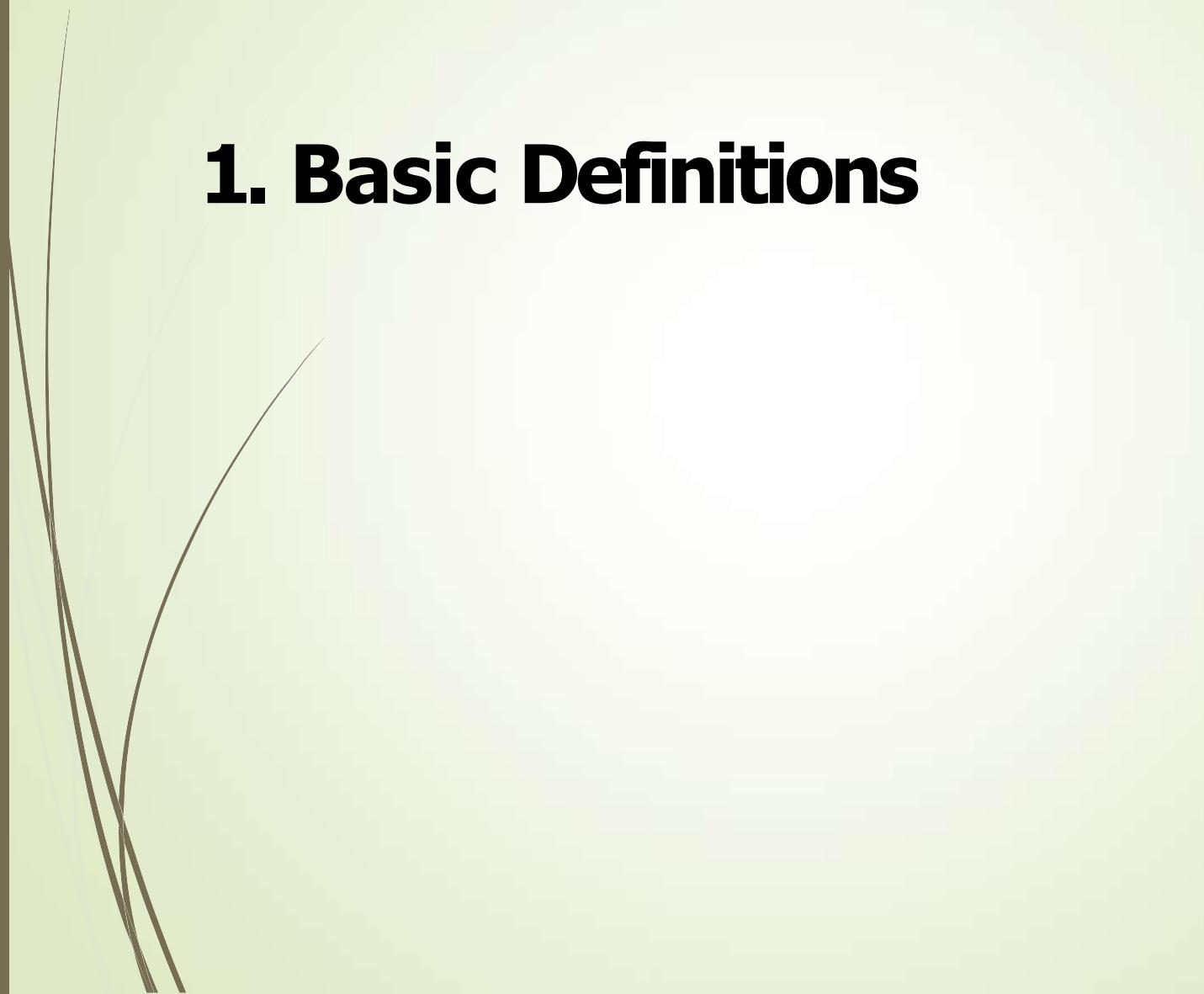


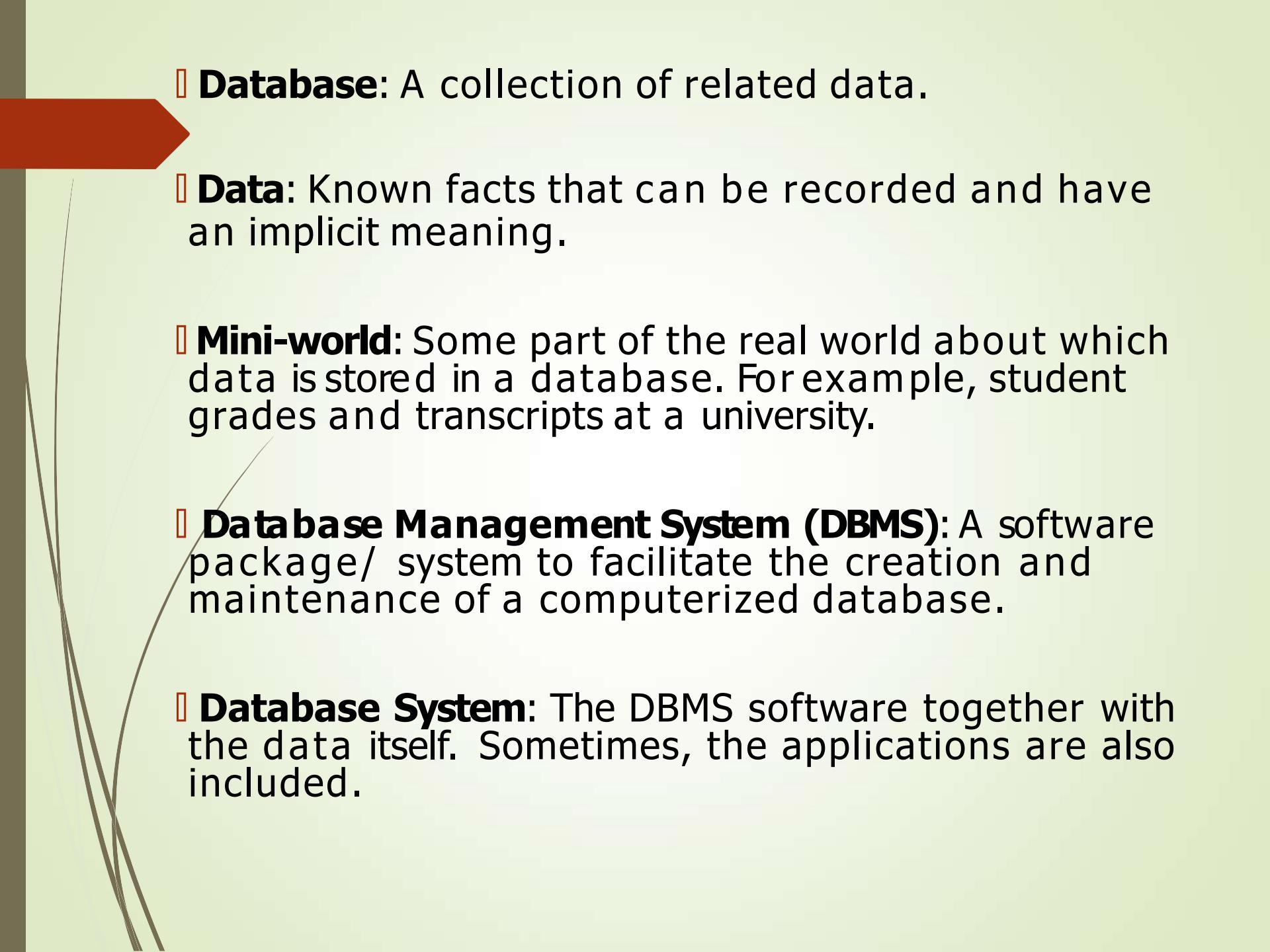
Topics

- | **Schemas versus Instances**
- | **Three-Schema Architecture**
- | **Data Independence**
- | **DBMS Languages**
- | **DBMS Interfaces**
- | **DBMS Component Modules**
- | **Database System Utilities**
- | **Classification of DBMSs**



1. Basic Definitions



- 
- | **Database:** A collection of related data.
 - | **Data:** Known facts that can be recorded and have an implicit meaning.
 - | **Mini-world:** Some part of the real world about which data is stored in a database. For example, student grades and transcripts at a university.
 - | **Database Management System (DBMS):** A software package/ system to facilitate the creation and maintenance of a computerized database.
 - | **Database System:** The DBMS software together with the data itself. Sometimes, the applications are also included.

Alternate Definition

- | **Database:** An integrated collection of more-or-less permanent data.
- | **Database Management System (DBMS):** A software package/ system to facilitate the creation and maintenance of a computerized database.



Concerns of DBMS:

integrity

consistency

redundancy (it's bad, but “replication” is good!)

security



2. Example of a Database (with a Conceptual Data Model)





I Mini-world for the example: Part of a UNIVERSITY environment.

I Some mini-world *entities*:

- STUDENTS
- COURSEs
- SECTIONS (of COURSEs)
- (academic) DEPARTMENTS
- INSTRUCTORS



Some mini-world *relationships*:

- SECTIONS are *of* specific COURSES
- STUDENTS *take* SECTIONS
- COURSES *have* prerequisite COURSES
- INSTRUCTORS *teach* SECTIONS
- COURSES *are offered by* DEPARTMENTS
- STUDENTS *major in* DEPARTMENTS

NOTE: The above could be expressed in the *ENTITY RELATIONSHIP* data model.

Figure 1.1 A simplified database system environment, illustrating the concepts and terminology discussed in Section 1.1.

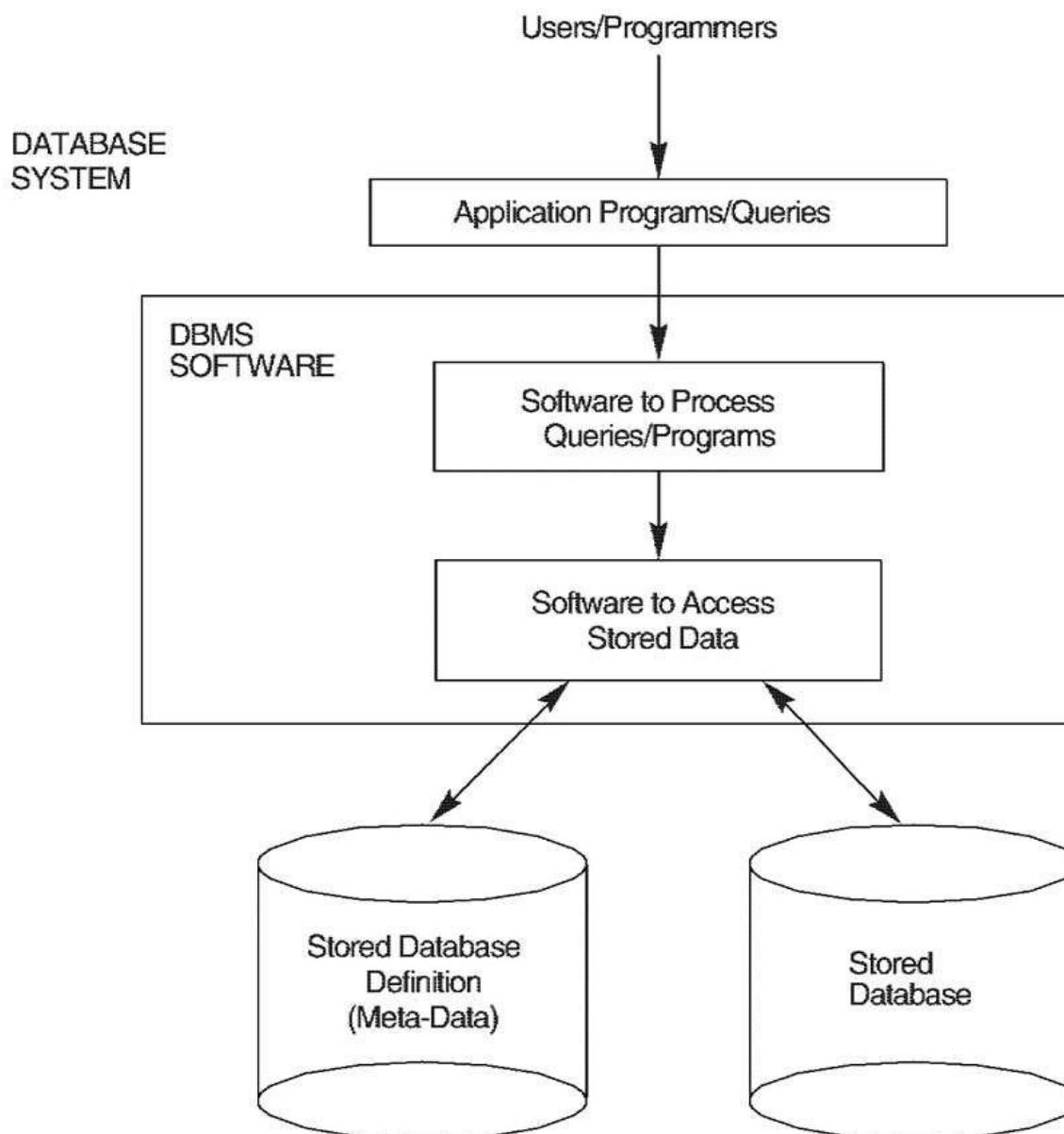


Figure 1.2 An example of a database that stores student records and their grades.

STUDENT	Name	StudentNumber	Class	Major
	Smith	17	1	CS
	Brown	8	2	CS

COURSE	CourseName	CourseNumber	CreditHours	Department
	Intro to Computer Science	CS1310	4	CS
	Data Structures	CS3320	4	CS
	Discrete Mathematics	MATH2410	3	MATH
	Database	CS3380	3	CS

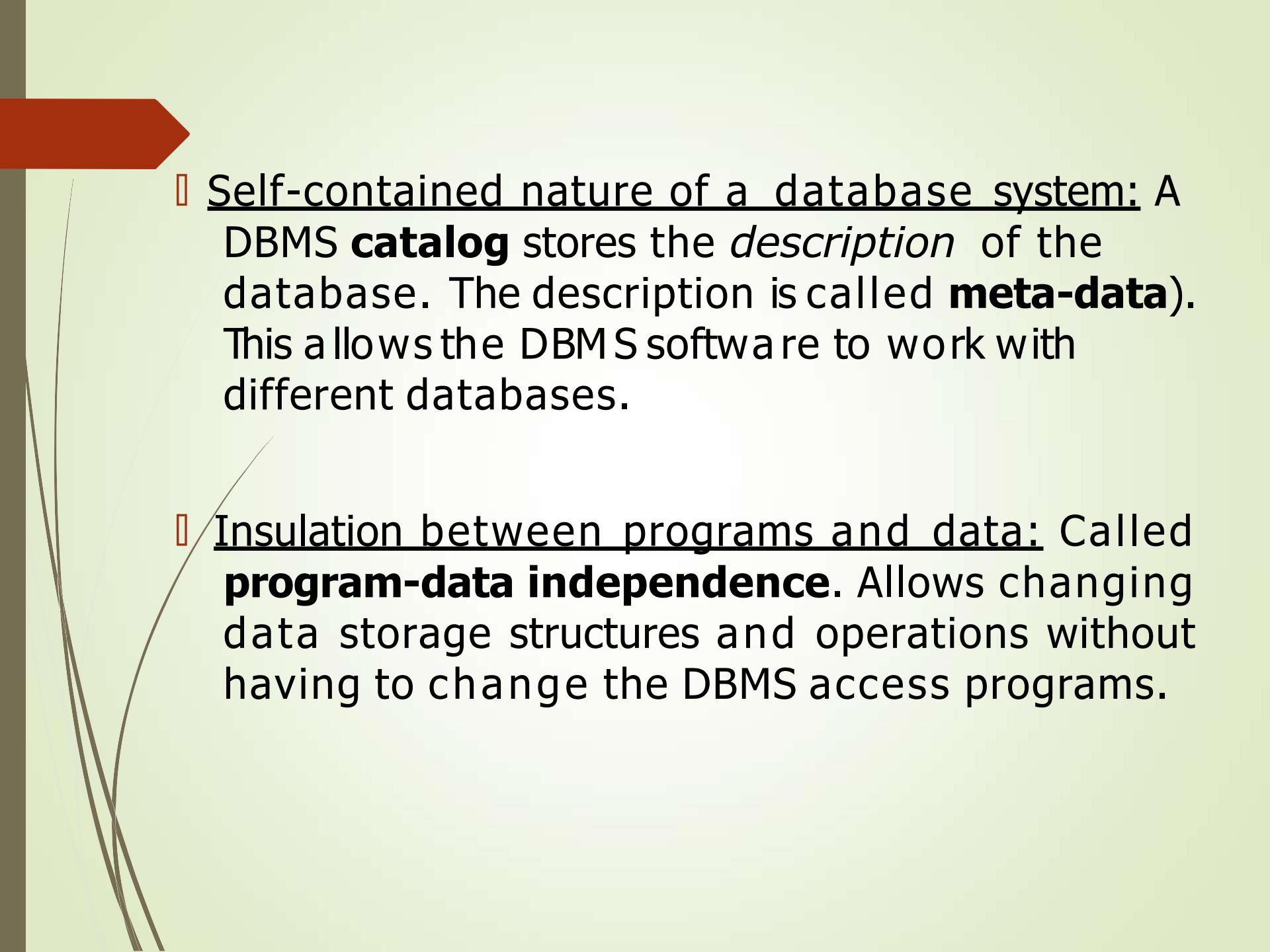
SECTION	SectionIdentifier	CourseNumber	Semester	Year	Instructor
	85	MATH2410	Fall	98	King
	92	CS1310	Fall	98	Anderson
	102	CS3320	Spring	99	Knuth
	112	MATH2410	Fall	99	Chang
	119	CS1310	Fall	99	Anderson
	135	CS3380	Fall	99	Stone

GRADE_REPORT	StudentNumber	SectionIdentifier	Grade
	17	112	B
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

PREREQUISITE	CourseNumber	PrerequisiteNumber
	CS3380	CS3320
	CS3380	MATH2410
	CS3320	CS1310



3. Main Characteristics of Database Technology

- 
- Self-contained nature of a database system: A DBMS **catalog** stores the *description* of the database. The description is called **meta-data**). This allows the DBMS software to work with different databases.
 - Insulation between programs and data: Called **program-data independence**. Allows changing data storage structures and operations without having to change the DBMS access programs.

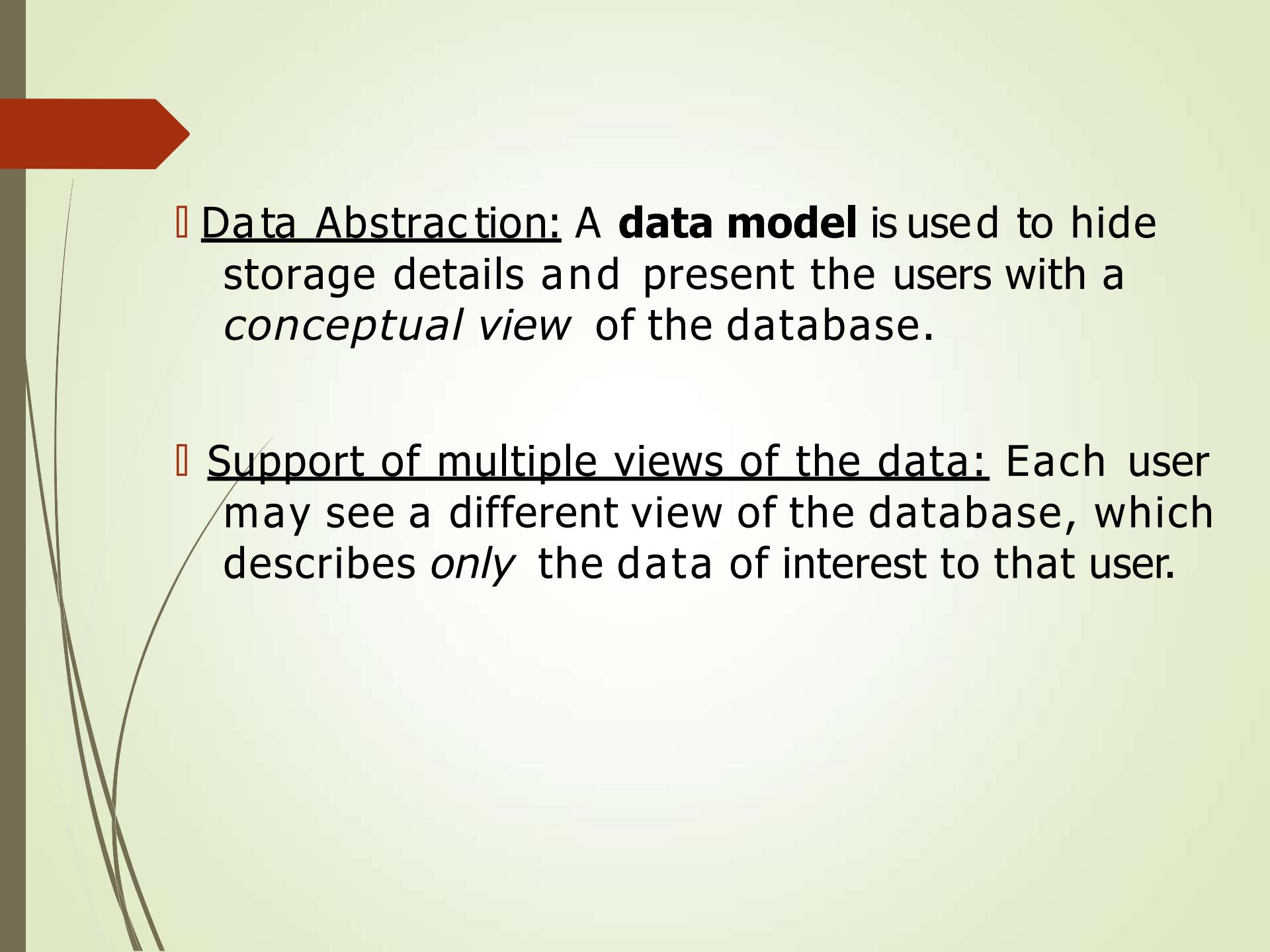
- 
- Data Abstraction: A **data model** is used to hide storage details and present the users with a *conceptual view* of the database.
 - Support of multiple views of the data: Each user may see a different view of the database, which describes *only* the data of interest to that user.

Figure 1.4 Two views derived from the example database shown in Figure 1.2. (a) The student transcript view. (b) The course prerequisite view.

(a)

TRANSCRIPT	StudentName	Student Transcript				
		CourseNumber	Grade	Semester	Year	SectionId
	Smith	CS1310	C	Fall	99	119
		MATH2410	B	Fall	99	112
	Brown	MATH2410	A	Fall	98	85
		CS1310	A	Fall	98	92
		CS3320	B	Spring	99	102
		CS3380	A	Fall	99	135

(b)

PREREQUISITES	CourseName	CourseNumber	Prerequisites
	Database	CS3380	CS3320
			MATH2410
	Data Structures	CS3320	CS1310



4. Additional Benefits of Database Technology



- | **Controlling redundancy in data storage and in development and maintenance efforts.**

- | **Sharing of data among multiple users.**

- | **Restricting unauthorized access to data.**

- | **Providing multiple interfaces to different classes of users.**

- | **Representing complex relationships among data.**

- | **Enforcing integrity constraints on the database.**



- | **Providing backup and recovery services.**

- | **Potential for enforcing standards.**

- | **Flexibility to change data structures.**

- | **Reduced application development time.**

- | **Availability of up-to-date information.**

- | **Economies of scale.**

Classes of Database Users

||(A) **Workers on the scene:** Persons whose job involves daily use of a large database.

Database administrators (DBAs)

Database designers

End users

Casual users

Parametric (naïve) users

Sophisticated end users

System analysts and application programmers



I (B) Workers behind the scene: Persons whose job involves design, development, operation and maintenance of the *DBMS software and system* environment.

DBMS designers and implementers

Tool developers

Operators and maintenance personnel

Figure 1.5 The redundant storage of data items. (a) *Controlled redundancy*: Including StudentName and CourseNumber in the grade_report file. (b) *Uncontrolled redundancy*: A GRADE_REPORT record that is inconsistent with the STUDENT records in Figure 1.2, because the Name of student number 17 is Smith, not Brown.

(a)

GRADE_REPORT	StudentNumber	StudentName	SectionIdentifier	CourseNumber	Grade
	17	Smith	112	MATH2410	B
	17	Smith	119	CS1310	C
	8	Brown	85	MATH2410	A
	8	Brown	92	CS1310	A
	8	Brown	102	CS3320	B
	8	Brown	135	CS3380	A

(b)

GRADE_REPORT	StudentNumber	StudentName	SectionIdentifier	CourseNumber	Grade
	17	Brown	112	MATH2410	B



5 When not to use a DBMS



■ **Main inhibitors (costs) of using a DBMS:**

- High initial investment and possible need for additional hardware.
- Overhead for providing generality, security, recovery, integrity, and concurrency control.

■ **When a DBMS may be unnecessary:**

- If the database and applications are simple, well defined, and not expected to change.
- If there are stringent real-time requirements that may not be met because of DBMS overhead.
- If access to data by multiple users is not required.

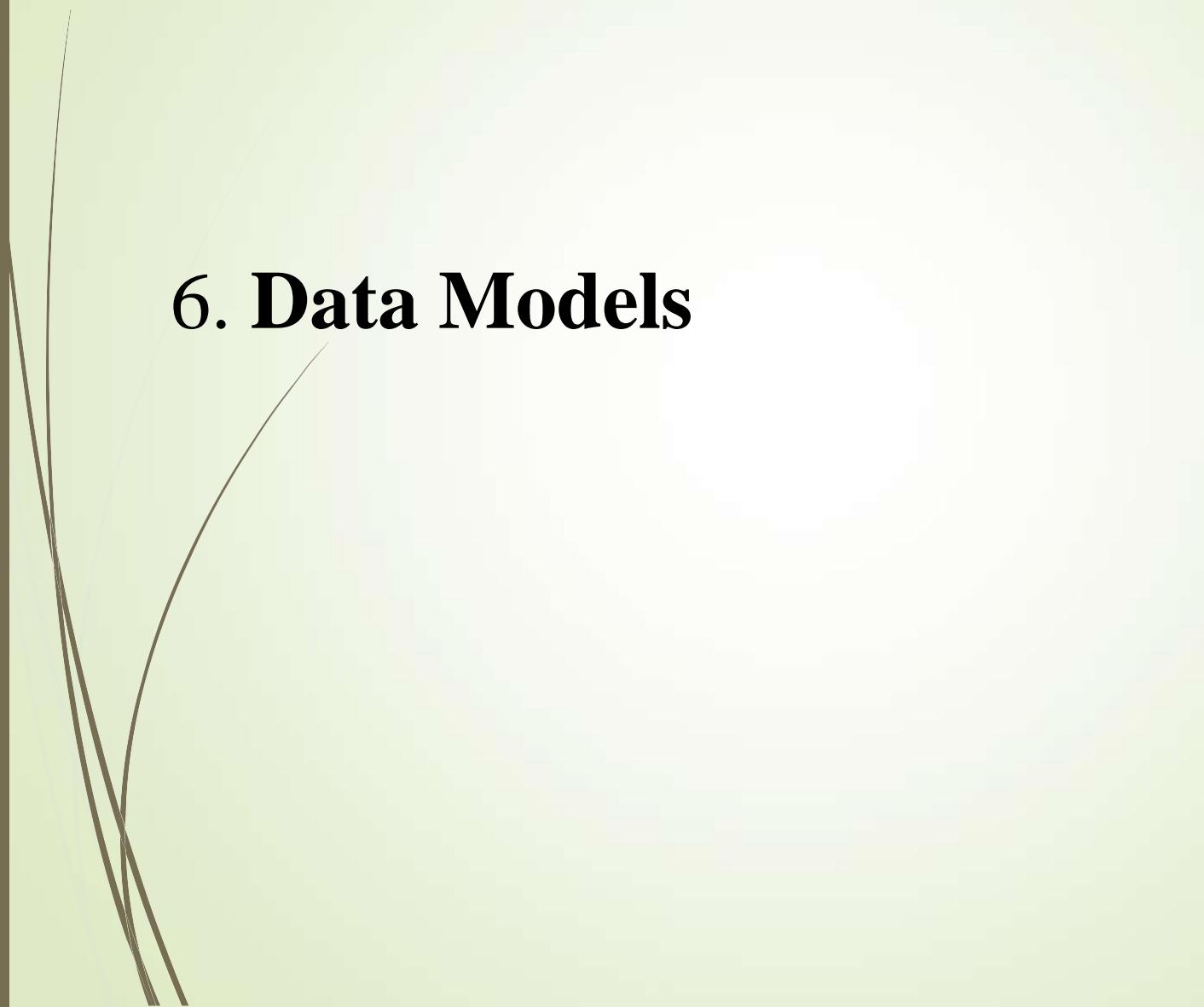


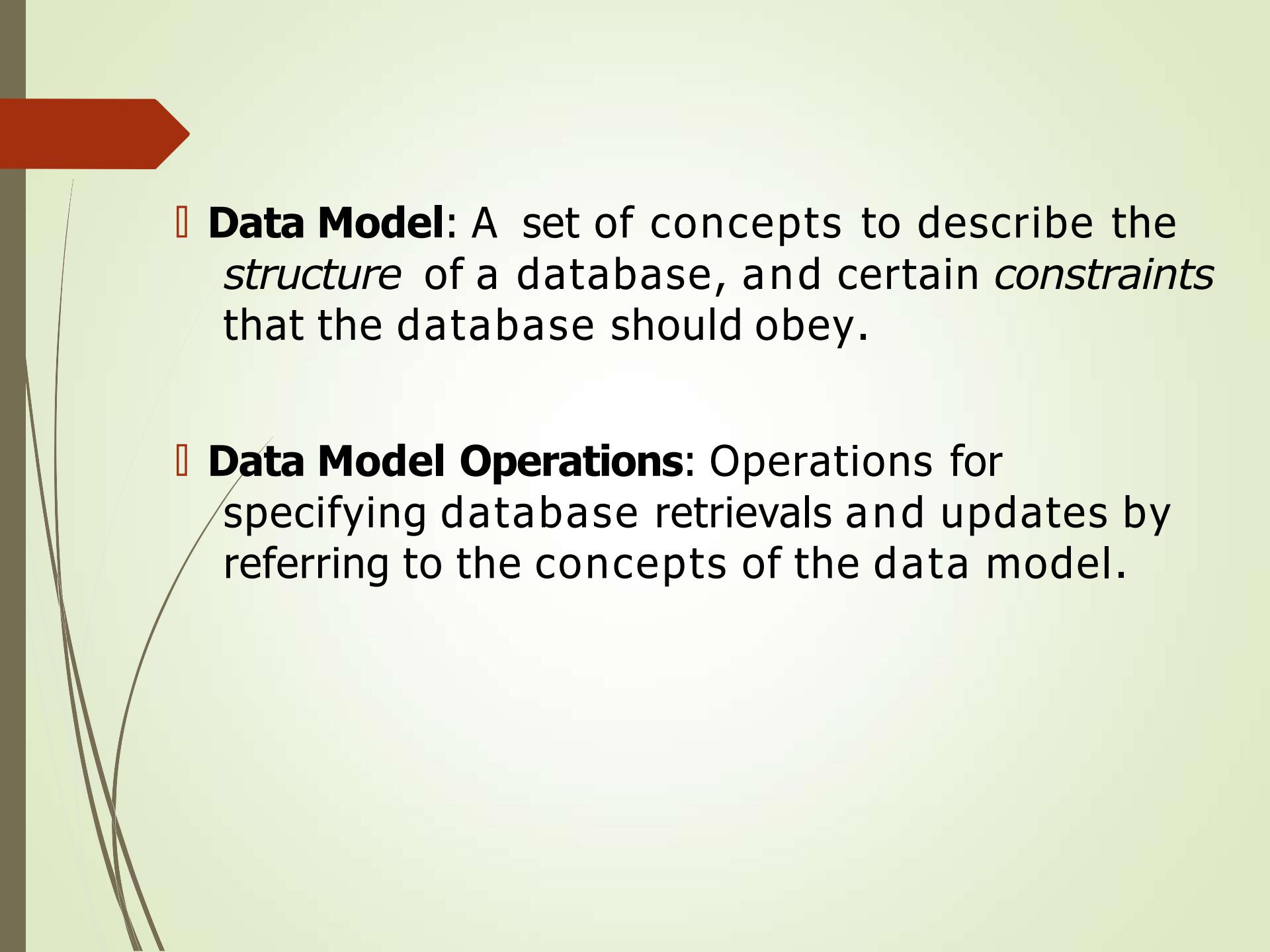
I **When no DBMS may suffice:**

- If the database system is not able to handle the complexity of data because of modeling limitations
- If the database users need special operations not supported by the DBMS.



6. Data Models



- 
- || **Data Model:** A set of concepts to describe the *structure* of a database, and certain *constraints* that the database should obey.
 - || **Data Model Operations:** Operations for specifying database retrievals and updates by referring to the concepts of the data model.



Categories of data models:

- || **Conceptual (high-level, semantic)** data models: Provide concepts that are close to the way many users *perceive* data. (Also called **entity-based** or **object-based** data models.)
- || **Physical (low-level, internal)** data models: Provide concepts that describe details of how data is stored in the computer.
- || **Implementation (record-oriented)** data models: Provide concepts that fall between the above two, balancing user views with some computer storage details.



6A. HISTORY OF DATA MODELS



- 
- || **Relational Model:** proposed in 1970 by E.F. Codd (IBM), first commercial system in 1981-82. Now in several commercial products (ORACLE, SYBASE, INFORMIX, CA -INGRES).
 - || **Network Model:** the first one to be implemented by Honeywell in 1964-65 (IDS System). Adopted heavily due to the support by CODASYL (CODASYL - DBTG report of 1971). Later implemented in a large variety of systems - IDMS (Cullinet - now CA), DMS 1100 (Unisys), IMAGE (H.P.), VAX -DBMS (Digital).
 - || **Hierarchical Data Model :** implemented in a joint effort by IBM and North American Rockwell around 1965. Resulted in the IMS family of systems. The most popular model. Other system based on this model: System 2k (SAS inc.)

- 
- I **Object-oriented Data Model(s)** : several models have been proposed for implementing in a database system. One set comprises models of persistent O-O Programming Languages such as C++ (e.g., in OBJECTSTORE or VERSANT), and Smalltalk (e.g., in GEMSTONE). Additionally, systems like O2, ORION (at MCC - then ITASCA), IRIS (at H.P.- used in Open OODB).
 - **Object-Relational Models** : Most Recent Trend. Exemplified in ILLUSTRA and UNiSQL systems.

Figure 2.1 Schema diagram for the database of Figure 1.2.

STUDENT

Name	StudentNumber	Class	Major
------	---------------	-------	-------

COURSE

CourseName	CourseNumber	CreditHours	Department
------------	--------------	-------------	------------

PREREQUISITE

CourseNumber	PrerequisiteNumber
--------------	--------------------

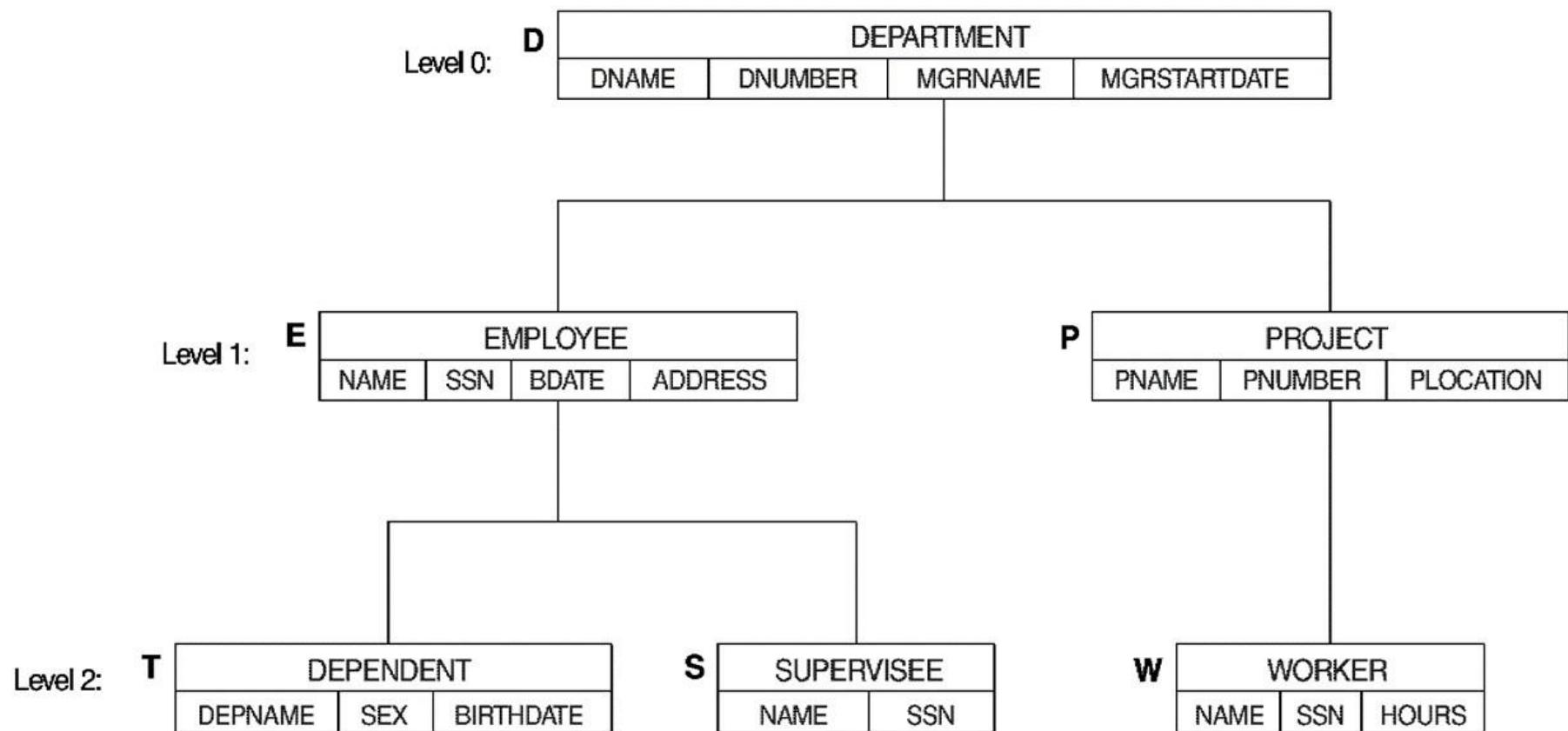
SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
-------------------	--------------	----------	------	------------

GRADE_REPORT

StudentNumber	SectionIdentifier	Grade
---------------	-------------------	-------

Figure D.4 A hierarchical schema for part of the COMPANY database.





6B. HIERARCHICAL MODEL



ADVANTAGES:

- I Hierarchical Model is simple to construct and operate on**
- I Corresponds to a number of natural hierarchically organized domains - e.g., assemblies in manufacturing, personnel organization in companies**
- I Language is simple; uses constructs like GET, GET UNIQUE, GET NEXT, GET NEXT WITHIN PARENT etc.**



DISADVANTAGES:

- I Navigational and procedural nature of processing**
- I Database is visualized as a linear arrangement of records**
 - . Little scope for "query optimization"**

Figure 2.4 The schema of Figure 2.1 in the notation of the network data model.

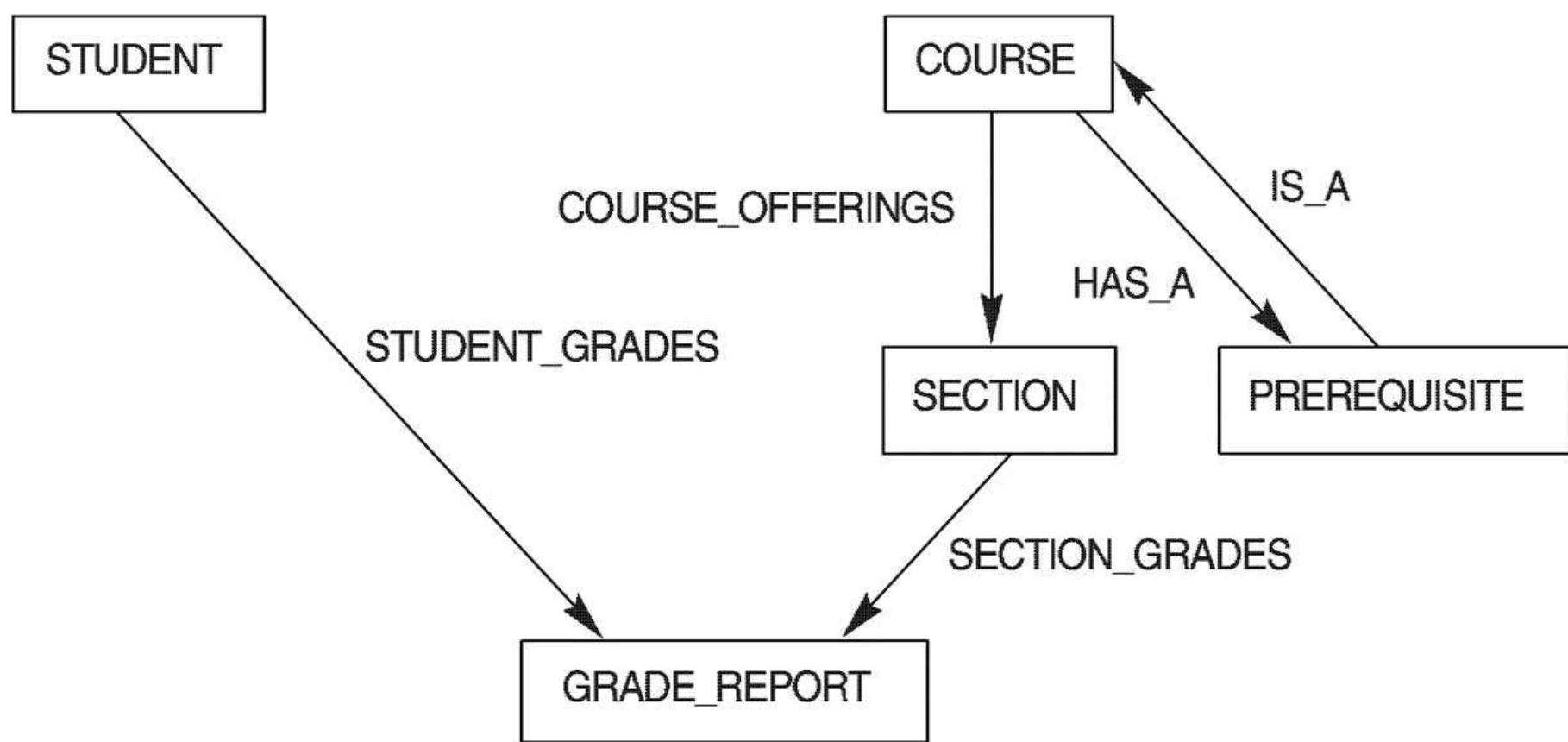
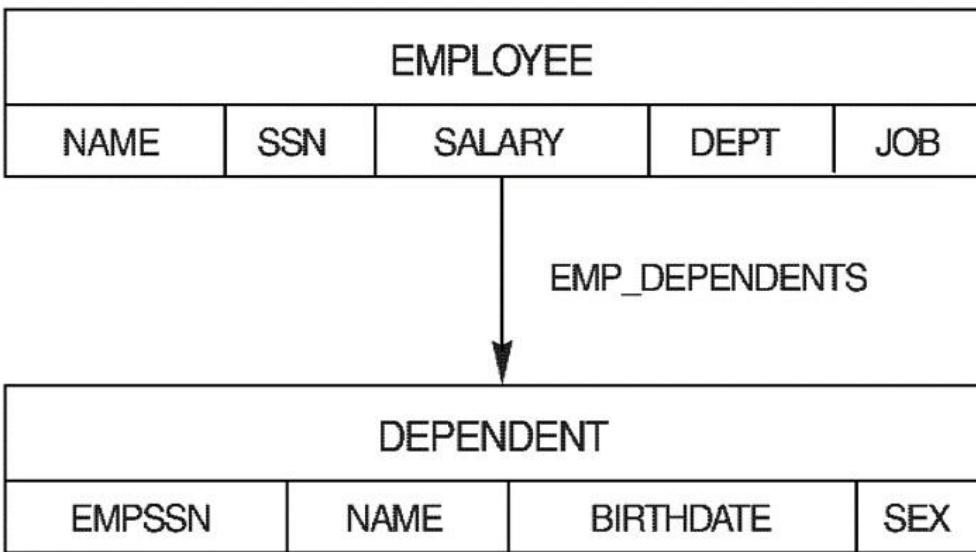
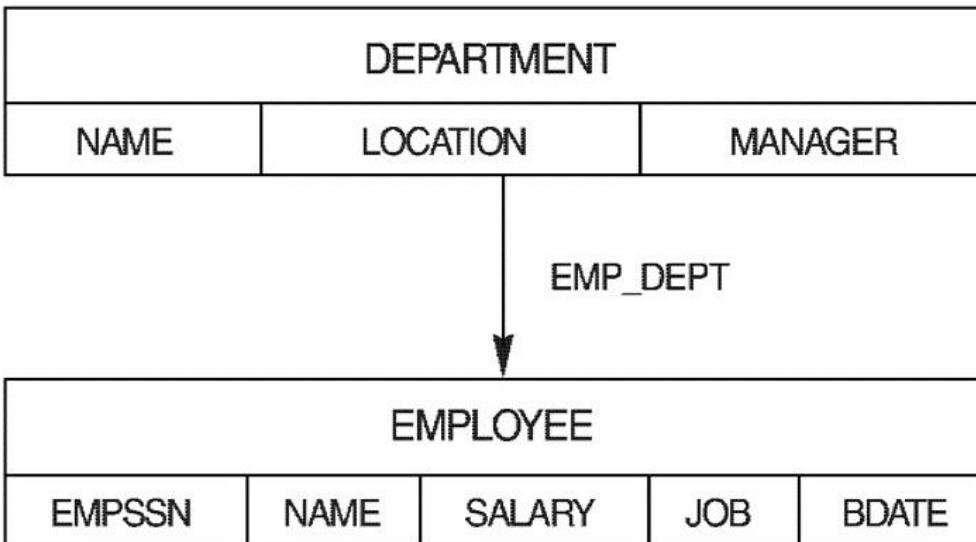


Figure C.7 Different set options. (a) An AUTOMATIC FIXED set.
(b) An AUTOMATIC MANDATORY set.

(a)



(b)





6C. NETWORK MODEL





ADVANTAGES:

- I Network Model is able to model complex relationships and represents semantics of add/delete on the relationships.**
- I Can handle most situations for modeling using record types and relationship types.**
- I Language is navigational; uses constructs like FIND, FIND member, FIND owner, FIND NEXT within set, GET etc. Programmers can do optimal navigation through the database.**



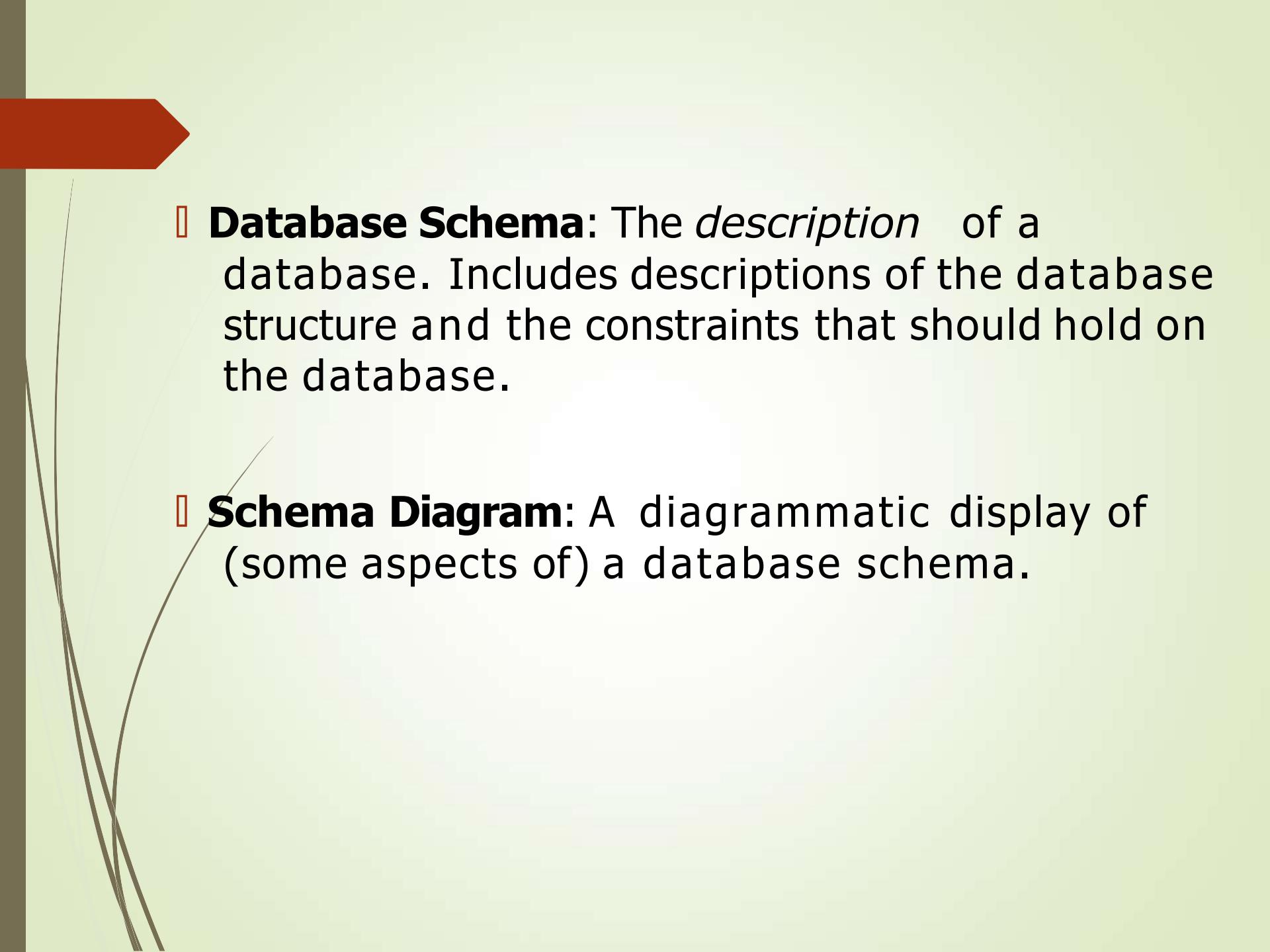
DISADVANTAGES:

- **Navigational and procedural nature of processing**
- **Database contains a complex array of pointers that thread through a set of records.**
- **Little scope for automated "query optimization"**



7. Schemas versus Instances

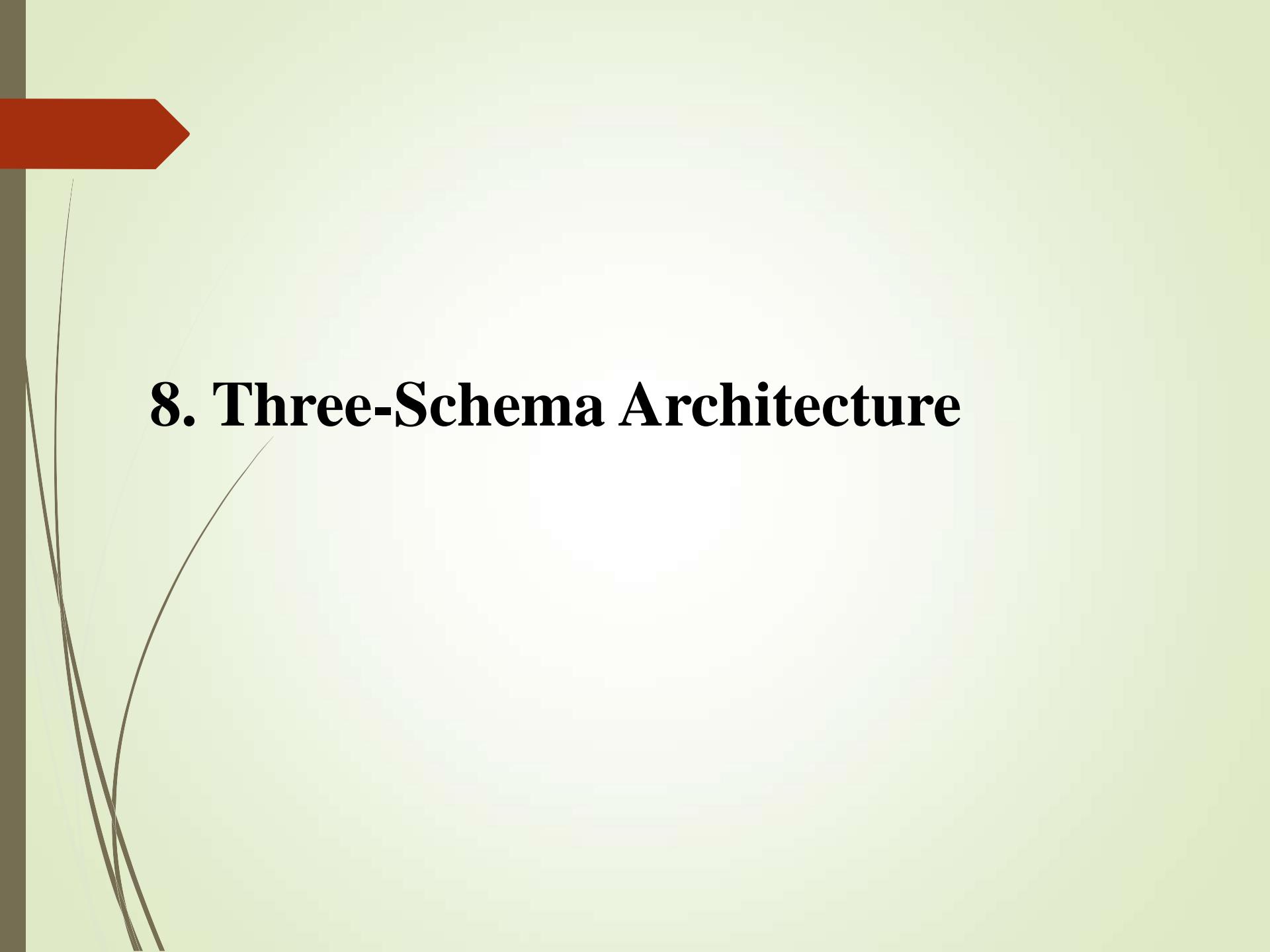


- 
- **Database Schema:** The *description* of a database. Includes descriptions of the database structure and the constraints that should hold on the database.
 - **Schema Diagram:** A diagrammatic display of (some aspects of) a database schema.

- 
- || **Database Instance:** The actual data stored in a database at a *particular moment in time* . Also called **database state** (or **occurrence**).
 - || The **database schema** changes very *infrequently* . The **database state** changes every *time the database is updated* . **Schema** is also called **intension**, whereas **state** is called **extension**.



8. Three-Schema Architecture



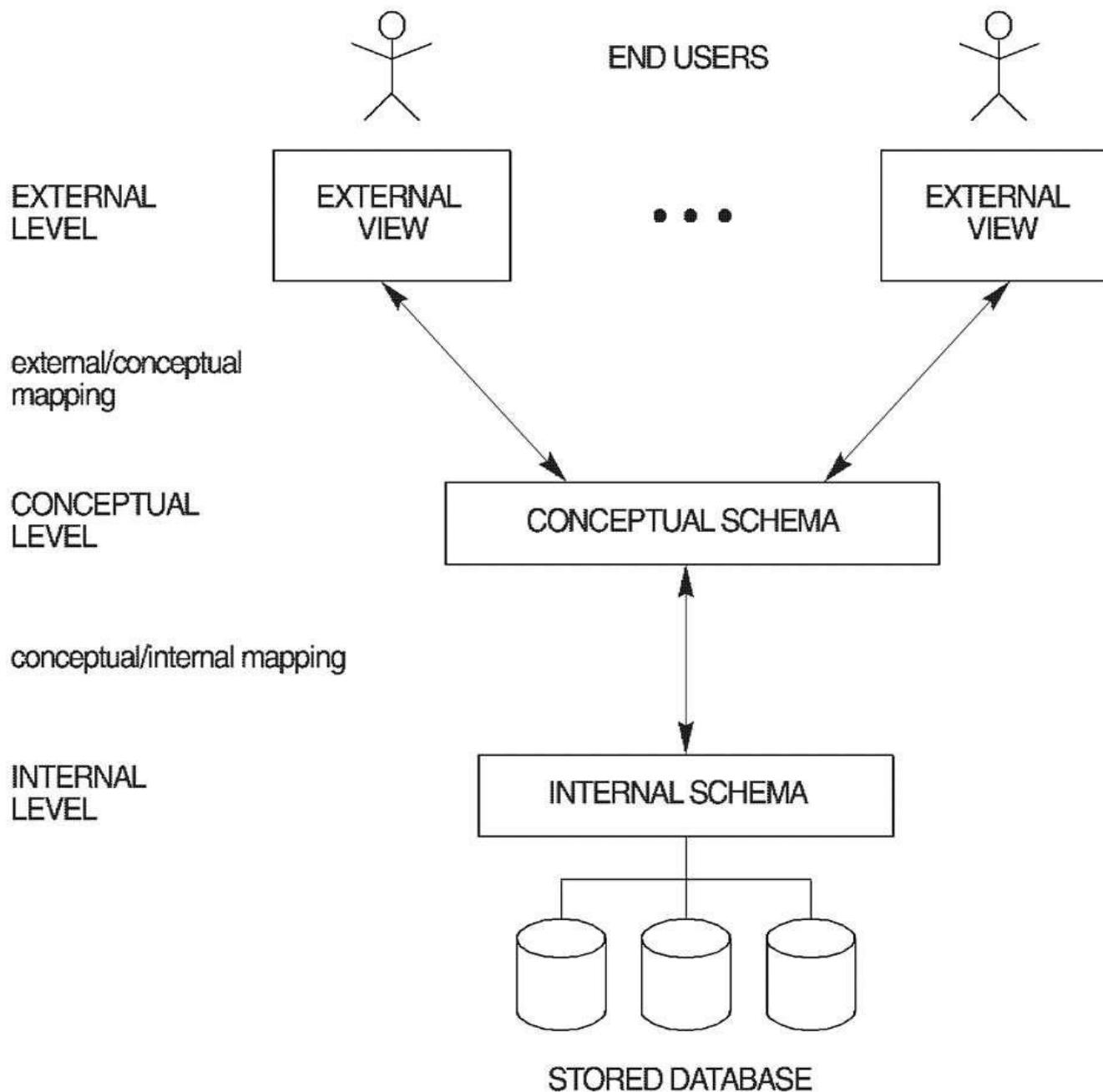
- 
- I Proposed to support DBMS characteristics of:
 - **Program-data independence.**
 - Support of **multiple views** of the data.

- 
- | Defines DBMS schemas at *three levels*:
 - Internal schema** at the internal level to describe data storage structures and access paths.
Typically uses a *physical* data model.
 - Conceptual schema** at the conceptual level to describe the structure and constraints for the *whole* database. Uses a *conceptual* or an *implementation* data model.
 - External schemas** at the external level to describe the various user views. Usually uses the same data model as the conceptual level.



Mappings among schema levels are also needed. Programs refer to an external schema, and are mapped by the DBMS to the internal schema forexecution.

Figure 2.2 Illustrating the three-schema architecture.





9 Data Independence



- **Logical Data Independence:** The capacity to change the conceptual schema without having to change the external schemas and their application programs.
- **Physical Data Independence:** The capacity to change the internal schema without having to change the conceptual schema.

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.



10. DBMS Languages



I Data Definition Language (DDL): Used by the DBA and database designers to specify the *conceptual schema* of a database. In many DBMSs, the DDL is also used to define internal and external schemas (views). In some DBMSs, separate **storage definition language (SDL)** and **view definition language (VDL)** are used to define internal and external schemas.



Data Manipulation Language (DML): Used to specify database retrievals and updates.

-DML commands (**data sublanguage**) can be *embedded* in a general-purpose programming language (**host language**), such as Java, C++, C, COBOL, PL/1 or PASCAL.

-Alternatively, *stand-alone* DML commands can be applied directly (**query language**).



11. DBMS Interfaces



- | **Stand-alone query language interfaces.**
- | **Programmer interfaces for embedding DML in programming languages:**
 - Pre-compiler Approach
 - Procedure (Subroutine) Call Approach
- | **User-friendly interfaces:**
 - Menu-based
 - Graphics-based (Point and Click, Drag and Drop etc.)
 - Forms-based
 - Natural language
 - Combinations of the above
 - Speech as Input (?) and Output
 - Web Browser as an interface

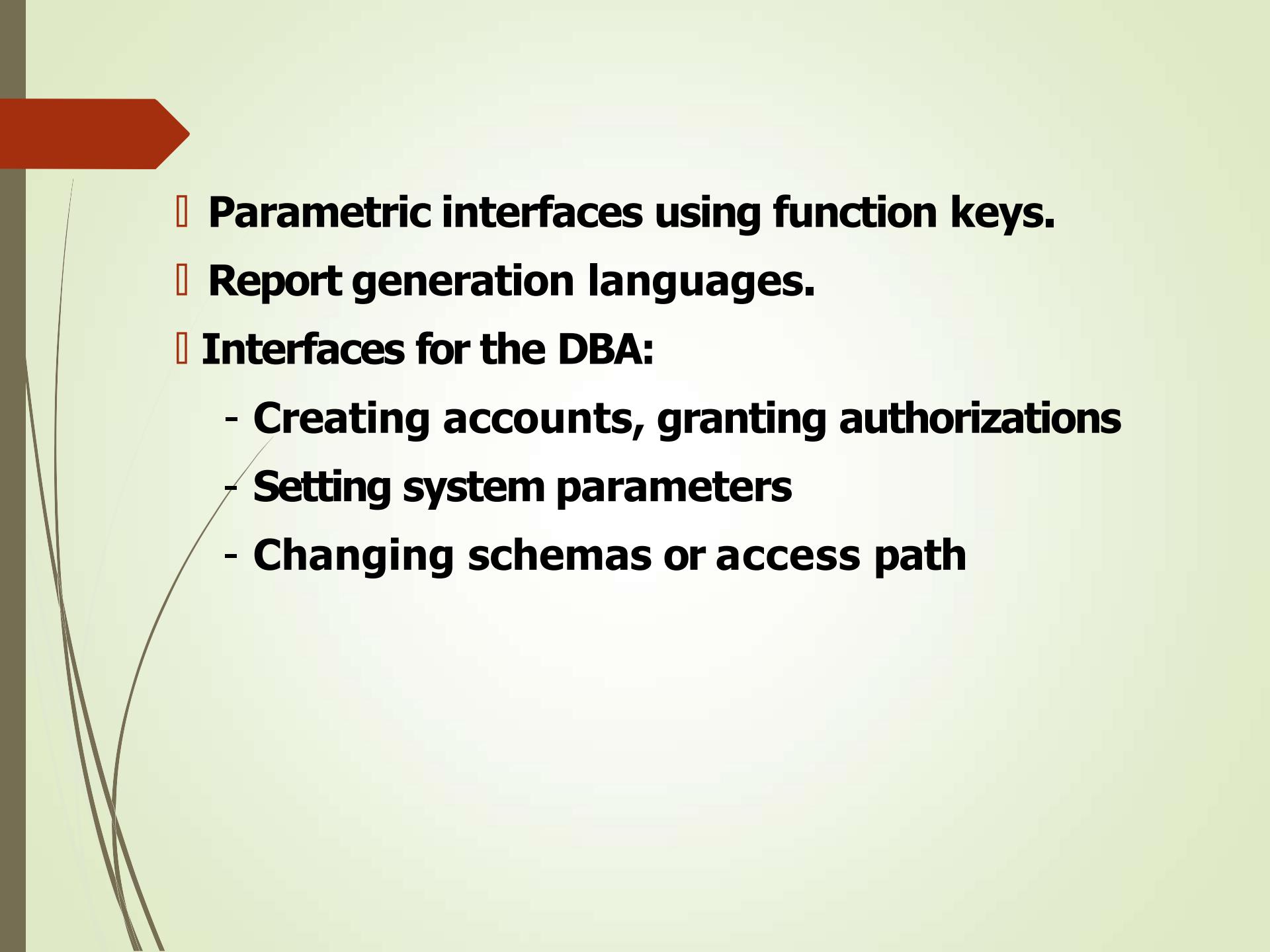
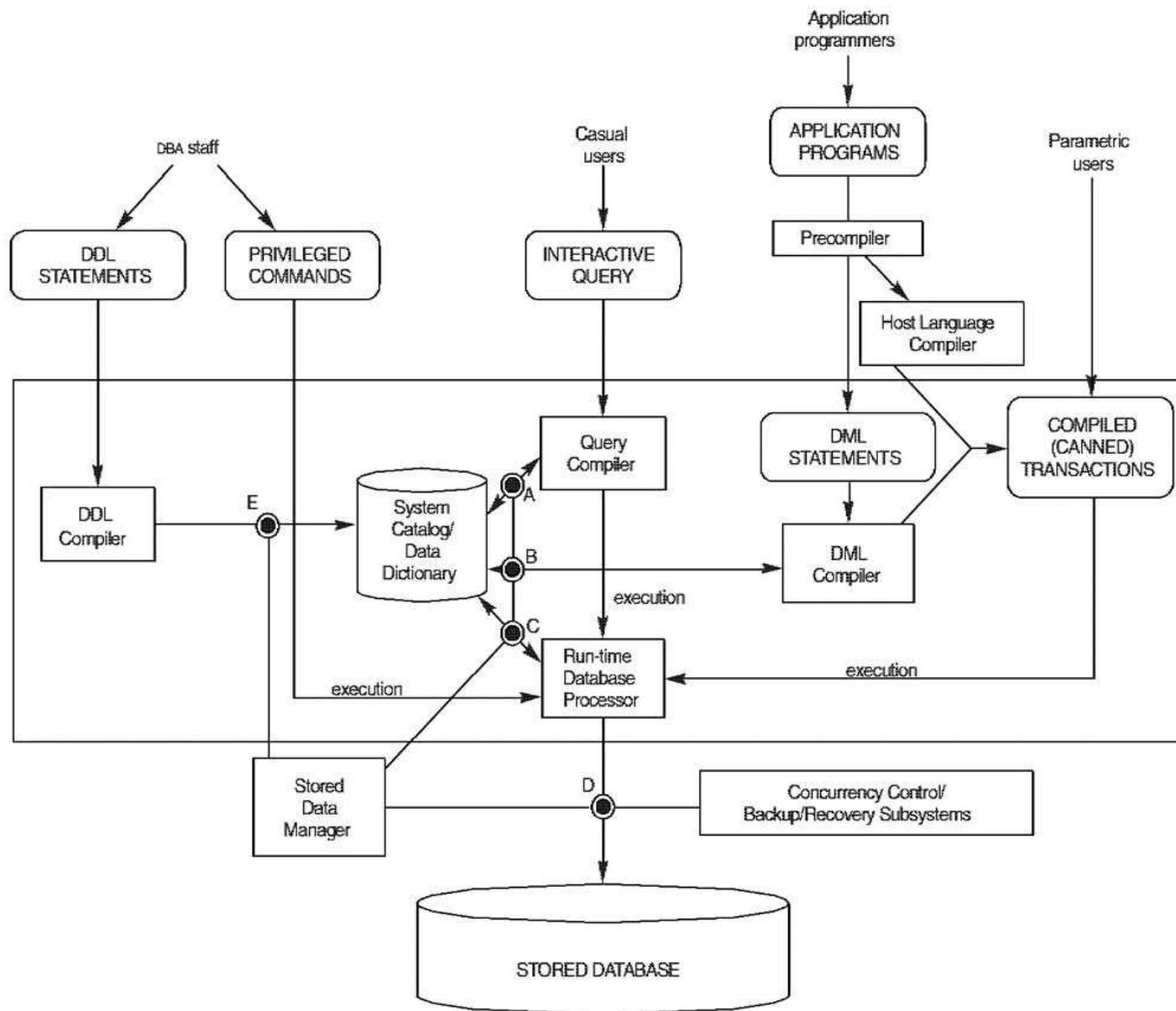
- 
- | **Parametric interfaces using function keys.**
 - | **Report generation languages.**
 - | **Interfaces for the DBA:**
 - **Creating accounts, granting authorizations**
 - **Setting system parameters**
 - **Changing schemas or access path**

Figure 2.3 Typical component modules of a DBMS. Dotted lines show accesses that are under the control of the stored data manager.





13. Database System Utilities





To perform certain functions such as:

- ***Loading* data stored in files into a database.**
- ***Backing up* the database periodically on tape.**
- ***Reorganizing* database file structures.**
- ***Report generation* utilities.**
- ***Performance monitoring* utilities.**
- **Other functions, such as *sorting* , *user monitoring* , *data compression* , etc.**



I **Data dictionary / repository:**

- Used to store schema descriptions and other information such as design decisions, application program descriptions, user information, usage standards, etc.
- Active** data dictionary is accessed by DBMS software and users/DBA.
- Passive** data dictionary is accessed by users/DBA only.



14. Classification of DBMSs



I Based on the data model used:

- **Traditional: Relational, Network, Hierarchical.**
- **Emerging: Object-oriented, Object-relational**

I Other classifications:

- **Single-user (typically used with micro-computers) vs. multi-user (most DBMSs).**
- **Centralized (uses a single computer with one database) vs. distributed (uses multiple computers, multiple databases)**

Distributed Database Systems have now come to be known as client server based database systems because they do not support a totally distributed environment, but rather a set of database servers supporting a set of clients.

Chapter 2

Database System Concepts
and Architecture

1

Outline

- Data Models and Their Categories
- Three-Schema Architecture
- Data Independence
- DBMS Languages and Interfaces
- Database System Utilities and Tools
- Centralized and Client-Server Architectures
- Classification of DBMSs

Data Models

|| Data Model:

|| A set of concepts to describe the ***structure*** of a database, the ***operations*** for manipulating these structures, and certain ***constraints*** that the database should obey.

|| Data Model Structure and Constraints:

|| Constructs are used to define the database structure

|| Constructs typically include ***elements***

Data Models (continued)

|| Data Model Operations:

- || These operations are used for specifying database *retrievals* and *updates* by referring to the constructs of the data model.
- || Operations on the data model may include ***basic model operations*** (e.g. generic insert, delete, update) and ***user-defined operations*** (e.g. compute_student_gpa, update_inventory)

Database Schema vs. Database State

- Database State:
 - Refers to the **content** of a database at a moment in time.
- Initial Database State:
 - Refers to the database state when it is initially loaded into the system.
- Valid State:
 - A state that satisfies the structure and constraints of the database.

Database Schema vs. Database State (continued)

Distinction

- The **database schema** changes very infrequently.
 - The **database state** changes every time the database is updated.
-
- **Schema** is also called **intension**.
 - **State** is also called **extension**.

Example of a Database Schema

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

Figure 2.1

Schema diagram for the database in Figure 1.2.

Example of a database state

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	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	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.

Three-Schema Architecture

- Proposed to support DBMS characteristics of:
 - **Program-data independence.**
 - Support of **multiple views** of the data .
- Not explicitly used in commercial DBMS products, but has been useful in explaining database system organization

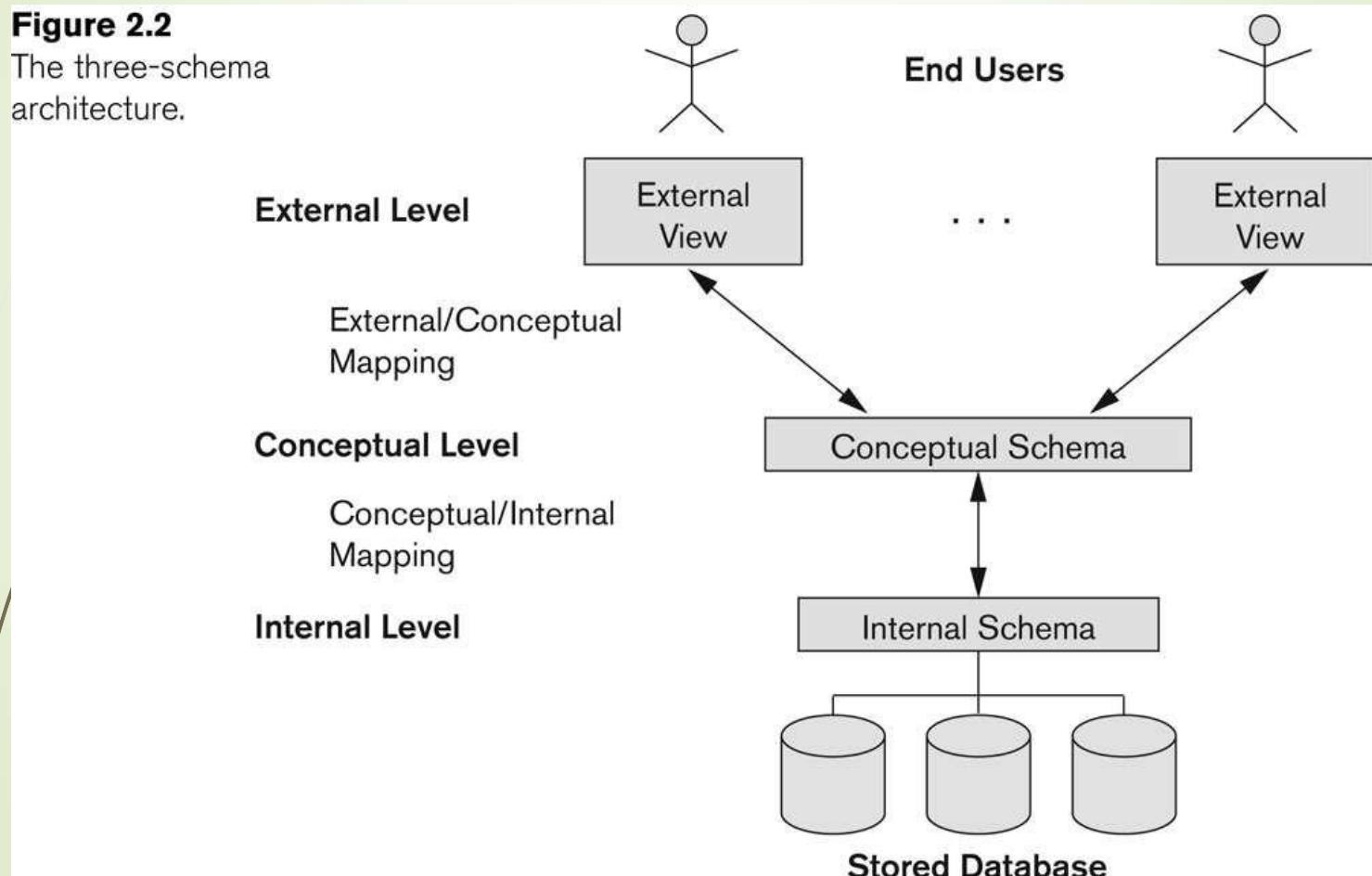
Three-Schema Architecture

- || Defines DBMS schemas at ***three*** levels:
 - || **Internal schema** at the internal level to describe physical storage structures and access paths (e.g indexes).
 - || Typically uses a **physical** data model.
 - || **Conceptual schema** at the conceptual level to describe the structure and constraints for the whole database for a community of users.
 - || Uses a **conceptual** or an **implementation**

The three-schema architecture

Figure 2.2

The three-schema architecture.



Three-Schema Architecture

- Mappings among schema levels are needed to transform requests and data.
 - Programs refer to an external schema, and are mapped by the DBMS to the internal schema for execution.
 - Data extracted from the internal DBMS level is reformatted to match the user's external view (e.g. formatting the results of an SQL query for display in a Web page)

Data Independence

|| **Logical Data Independence:**

- || The capacity to change the conceptual schema without having to change the external schemas and their associated application programs.

|| **Physical Data Independence:**

- || The capacity to change the internal schema without having to change the conceptual schema.
- || 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 (continued)

- 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.

DBMS Languages

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
 - High-Level or Non-procedural Languages: These include the relational language SQL
 - May be used in a standalone way or may be embedded in a programming language
 - Low Level or Procedural Languages:
 - These must be embedded in a programming language

DBMS Languages

■ Data Definition Language (DDL):

- Used by the DBA and database designers to specify the conceptual schema of a database.
- In many DBMSs, the DDL is also used to define internal and external schemas (views).
- In some DBMSs, separate **storage definition language (SDL)** and **view definition language (VDL)** are used to define internal and external schemas.
 - SDL is typically realized via DBMS commands provided to the DBA and database designers

DBMS Languages

|| Data Manipulation Language (DML):

- || Used to specify database retrievals and updates
- || DML commands (data sublanguage) can be *embedded* in a general-purpose programming language (host language), such as COBOL, C, C++, or Java.
- || A library of functions can also be provided to access the DBMS from a programming language
- || Alternatively, stand-alone DML commands can be applied directly (called a *query language*).

Types of DML

- **High Level or Non-procedural Language:**
 - For example, the SQL relational language
 - Are “set”-oriented and specify what data to retrieve rather than how to retrieve it.
 - Also called **declarative** languages.
- **Low Level or Procedural Language:**
 - Retrieve data one record-at-a-time;
 - Constructs such as looping are needed to retrieve multiple records, along with positioning pointers.

DBMS Interfaces

- Stand-alone query language interfaces
 - Example: Entering SQL queries at the DBMS interactive SQL interface (e.g. SQL*Plus in ORACLE)
- Programmer interfaces for embedding DML in programming languages
- User-friendly interfaces
 - Menu-based, forms-based, graphics-based, etc.

DBMS Programming Language Interfaces

- Programmer interfaces for embedding DML in a programming languages:
 - **Embedded Approach:** e.g embedded SQL (for C, C++, etc.), SQLJ (for Java)
 - **Procedure Call Approach:** e.g. JDBC for Java, ODBC for other programming languages
 - **Database Programming Language Approach:** e.g. ORACLE has PL/SQL, a programming language based on SQL; language incorporates SQL and its data types as integral components

User-Friendly DBMS Interfaces

- Menu-based, popular for browsing on the web
- Forms-based, designed for naïve users
- Graphics-based
 - (Point and Click, Drag and Drop, etc.)
- Natural language: requests in written English
- Combinations of the above:
 - For example, both menus and forms used extensively in Web database interfaces

Other DBMS Interfaces

- Speech as Input and Output
- Web Browser as an interface
- Parametric interfaces, e.g., bank tellers using function keys.
- Interfaces for the DBA:
 - Creating user accounts, granting authorizations
 - Setting system parameters
 - Changing schemas or access paths

Database System Utilities

- To perform certain functions such as:
 - Loading data stored in files into a database. Includes data conversion tools.
 - Backing up the database periodically on tape.
 - Reorganizing database file structures.
 - Report generation utilities.
 - Performance monitoring utilities.
 - Other functions, such as sorting, user monitoring, data compression, etc.

Centralized and Client-Server DBMS Architectures

I Centralized DBMS:

- Combines everything into single system including- DBMS software, hardware, application programs, and user interface processing software.
- User can still connect through a remote terminal – however, all processing is done at centralized site.

A Physical Centralized Architecture

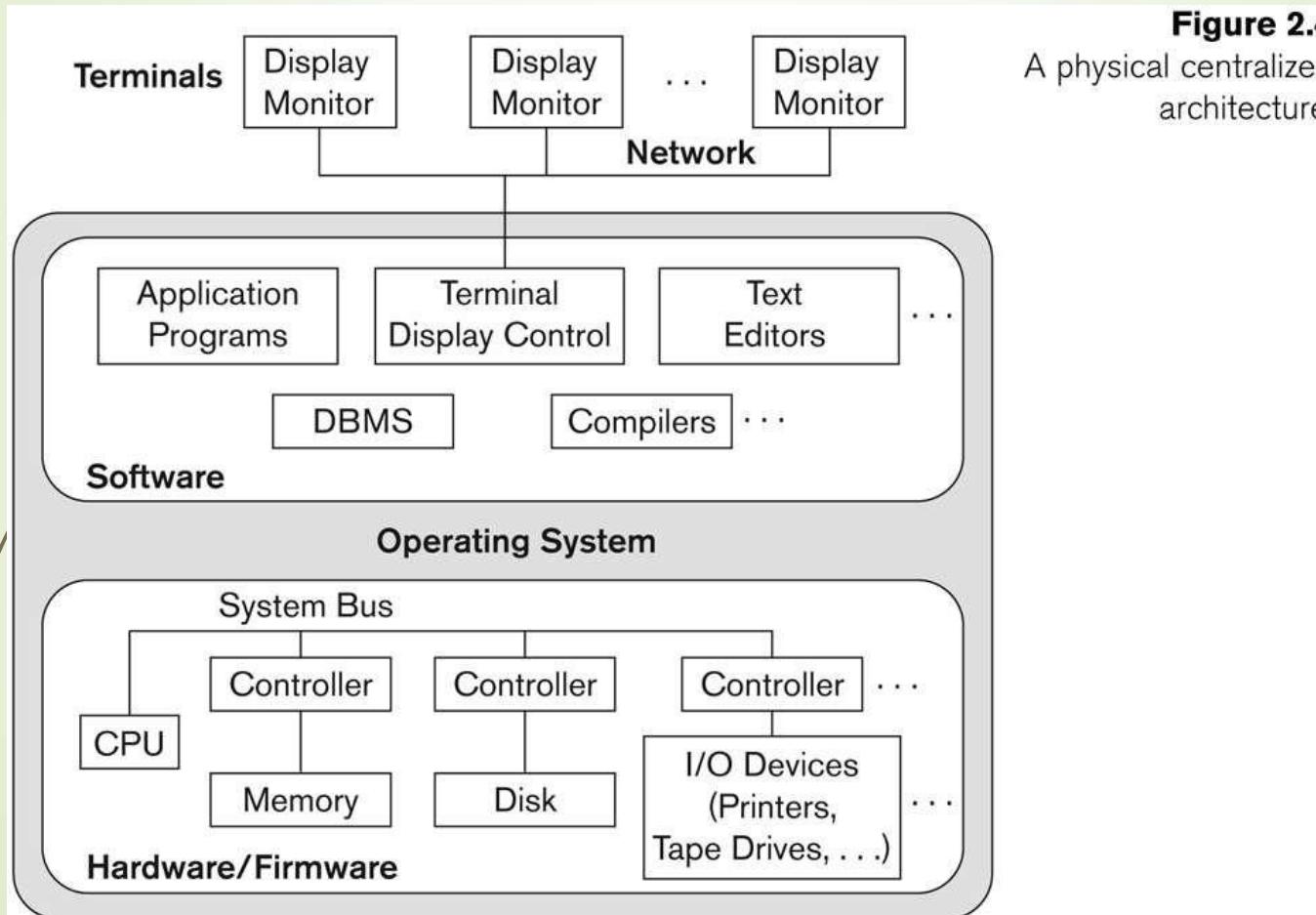


Figure 2.4

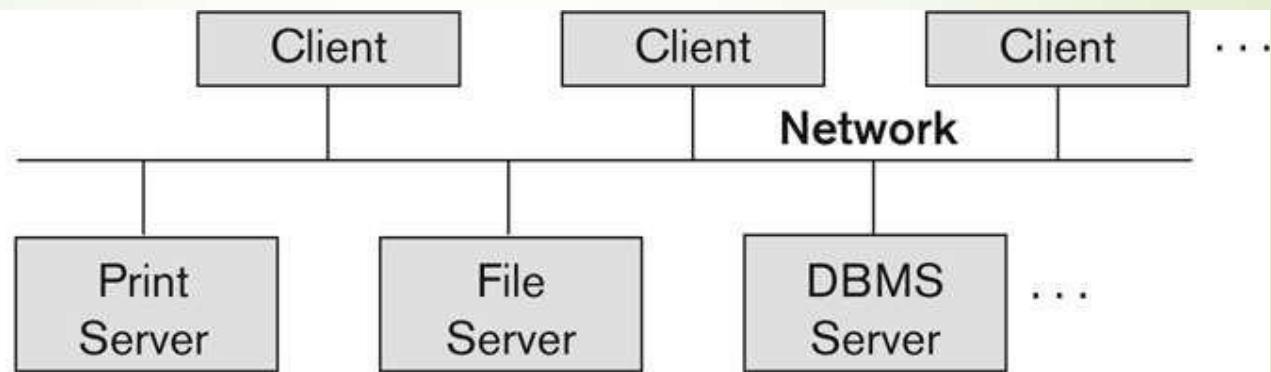
A physical centralized architecture.

Basic 2-tier Client-Server Architectures

- Specialized Servers with Specialized functions
 - Print server
 - File server
 - DBMS server
 - Web server
 - Email server
- Clients can access the specialized servers as needed

Logical two-tier client server architecture

Figure 2.5
Logical two-tier
client/server
architecture.



Clients

- Provide appropriate interfaces through a client software module to access and utilize the various server resources.
- Clients may be diskless machines or PCs or Workstations with disks with only the client software installed.
- Connected to the servers via some form of a network.
 - (LAN: local area network, wireless network, etc.)

DBMS Server

- || Provides database query and transaction services to the clients
- || Relational DBMS servers are often called SQL servers, query servers, or transaction servers
- || Applications running on clients utilize an Application Program Interface (**API**) to access server databases via standard interface such as:
 - || ODBC: Open Database Connectivity

Two Tier Client-Server Architecture

- A client program may connect to several DBMSs, sometimes called the data sources.
- In general, data sources can be files or other non-DBMS software that manages data.
- Other variations of clients are possible: e.g., in some object DBMSs, more functionality is transferred to clients including data dictionary functions, optimization and recovery across multiple servers, etc.

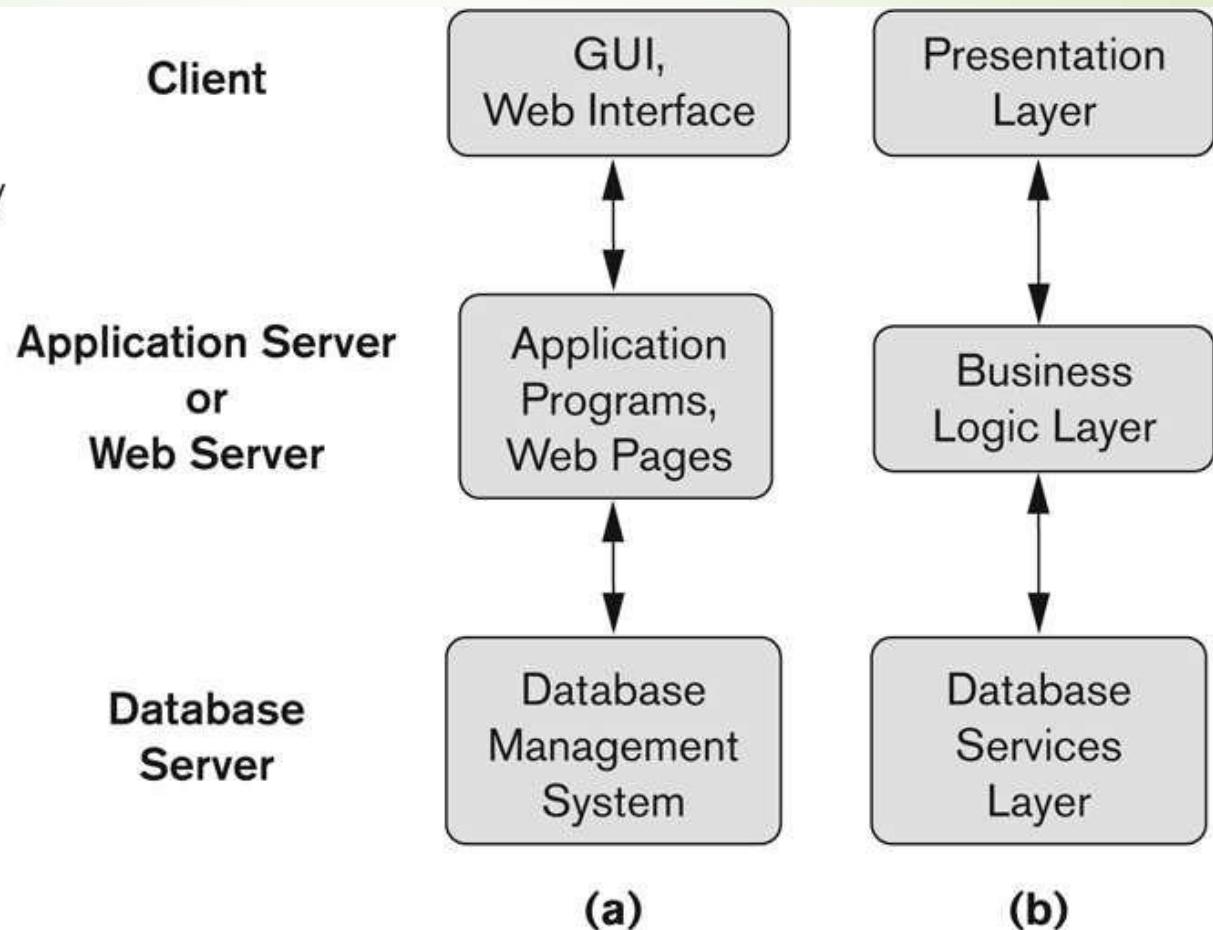
Three Tier Client-Server Architecture

- || Common for Web applications
- || Intermediate Layer called Application Server or Web Server:
 - || Stores the web connectivity software and the business logic part of the application used to access the corresponding data from the database server
 - || Acts like a conduit for sending partially processed data between the database server and the client.

Three-tier client-server architecture

Figure 2.7

Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

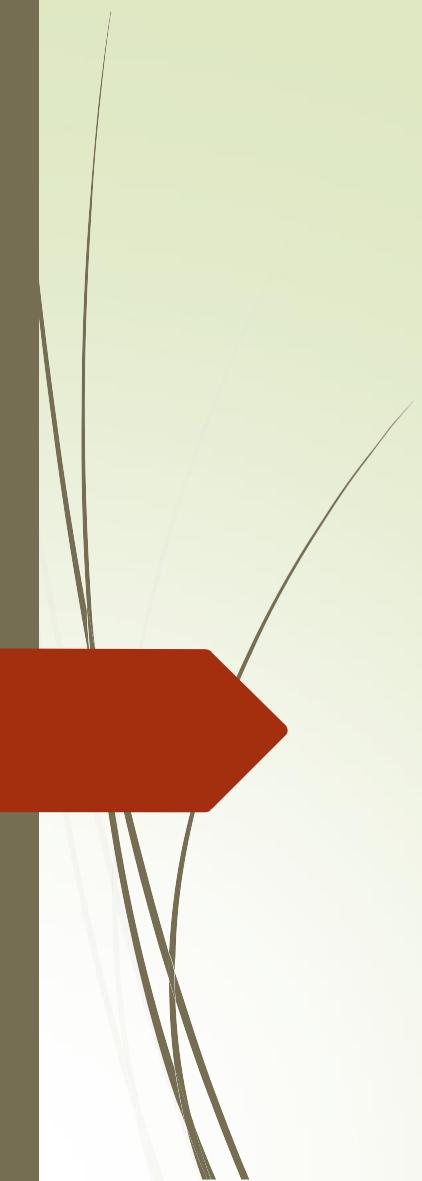


Cost considerations for DBMSs

- || Cost Range: from free open-source systems to configurations costing millions of dollars
- || Examples of free relational DBMSs: MySQL, PostgreSQL, others
- || Commercial DBMS offer additional specialized modules, e.g. time-series module, spatial data module, document module, XML module
 - || These offer additional specialized

Summary

- Data Models and Their Categories
- History of Data Models
- Schemas, Instances, and States
- Three-Schema Architecture
- Data Independence
- DBMS Languages and Interfaces
- Database System Utilities and Tools
- Centralized and Client-Server Architectures
- Classification of DBMSs



MODULE 2

Relational Model

Relational Model Concepts

Represents data as a collection of relations

➤ **Table of values**

- Row - Represents a collection of related data values

➤ Table name and column names

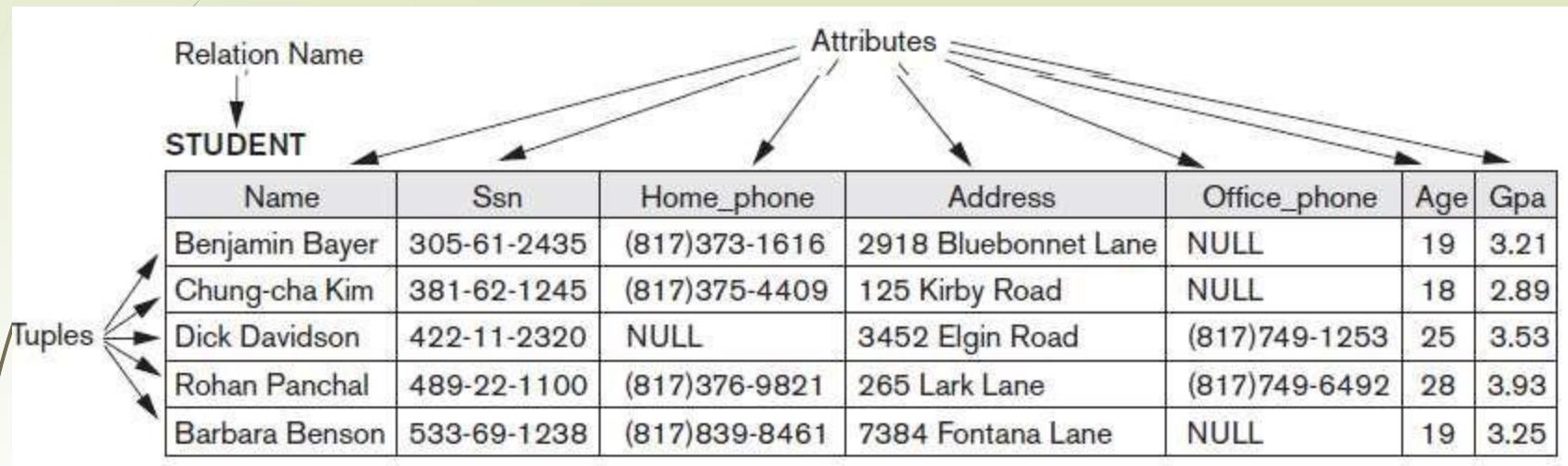
- Interpret the meaning of the values in each row

➤ For example: STUDENT table

- each row represents facts about a particular student entity.
- The column names—Name, Student_number, Class, and Major— specify how to interpret the data values in each row
- All values in a column are of the same data type.

➤ In the formal relational model terminology,

- a row is called a **tuple**
- a column header is called an **attribute**
- table is called a **relation**



Domains, Attributes, Tuples, and Relations

► Domain

- A **domain** D is a set of atomic values
- atomic - each value in the domain is indivisible
- 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
- Examples :
 - Usa_phone_numbers: The set of ten-digit phone numbers valid in the United States.
 - Names: The set of character strings that represent names of persons.
 - Social_security_numbers: The set of valid nine-digit Social

- A **data type** or **format** is also specified for each domain
- For example, the data type for the domain `Usa_phone_numbers` can be declared as a character string of the form $(ddd) \ ddd-dddd$, where each d is a numeric digit and the first three digits form a valid telephone area code.

➤ **Attribute**

- The *name* of the role played by some value (coming from some domain) in the context of a **relational schema**.
- The domain of attribute A is denoted $\text{dom}(A)$.

➤ Tuple

- mapping from attributes to values drawn from the respective domains of those attributes
- intended to describe some entity (or relationship between entities) in the miniworld
- Example: a tuple for a PERSON entity might be
 $\{ \text{Name} \rightarrow \text{"smith"}, \text{Gender} \rightarrow \text{Male}, \text{Age} \rightarrow 25 \}$

➤ Relation:

- A named set of tuples all of the same form i.e., having the same set of attributes

Relation schema

- used for describing the structure of a relation
- 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
- R is called the **name** of this relation
- 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 $\text{dom}(A_i)$
- The **degree (or arity)** of a relation is the number of attributes n of its relation schema

- **Example:** A relation of degree seven, which stores information about university students

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)

- domains for some of the attributes of the STUDENT relation:
 $\text{dom}(\text{Name}) = \text{Names}$; $\text{dom}(\text{Ssn}) = \text{Social_security_numbers}$;
 $\text{dom}(\text{HomePhone}) = \text{USA_phone_numbers}$,
 $\text{dom}(\text{Office_phone}) = \text{USA_phone_numbers}$,

Relation (or relation state)

- A **relation** (or **relation state**) r of the relation schema by $R(A_1, A_2, \dots, A_n)$, 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 $t. A_i$
- The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

Characteristics of Relations

1. Ordering of Tuples in a Relation

- Relation defined as a set of tuples
- Elements have no order among them
- a relation is not sensitive to the ordering of tuples

2. Ordering of Values within a Tuple and an Alternative Definition of a Relation

- Order of attributes and values is not that important
- As long as correspondence between attributes and values maintained
- An alternative definition of a relation can be given, making the ordering of values in a tuple unnecessary

- In this definition A **relation schema** $R(A_1, A_2, \dots, A_n)$, set 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
- 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 $\text{dom}(A_i)$
- The ordering of attributes is not important, because the attribute name appears with its value.

3. Values and NULLs in the Tuples

- Each value in a tuple is atomic
- NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple
- For example some STUDENT tuples have NULL for their office phones because they do not have an office
- Another student has a NULL for home phone
- 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**).

4. 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 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 particular instance of the assertion.
- | For example, the first tuple asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

Relational Model Notation

- Relation schema R of degree n is denoted by $R(A_1, A_2, \dots, A_n)$
- Uppercase letters Q, R, S denote relation names
- Lowercase letters q, r, s denote relation states
- 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
- 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.

Relational Model Constraints and Relational Database Schemas

Constraints

- Restrictions on the actual values in a database state
- Derived from the rules in the miniworld that the database represents
- Three main categories:

1. inherent model-based constraints or implicit constraints

- inherent in the data model

2. schema-based constraints or explicit constraints

- can be directly expressed in schemas of the data model

3. application-based or semantic constraints or business rules

- cannot be directly expressed in the schemas
- Expressed and enforced by application program

Domain Constraints

- specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$
- Typically include:
 - Numeric data types for integers and real numbers
 - Characters
 - Booleans
 - Date, time, timestamp
 - Other special data types

Key Constraints and Constraints on NULL Values

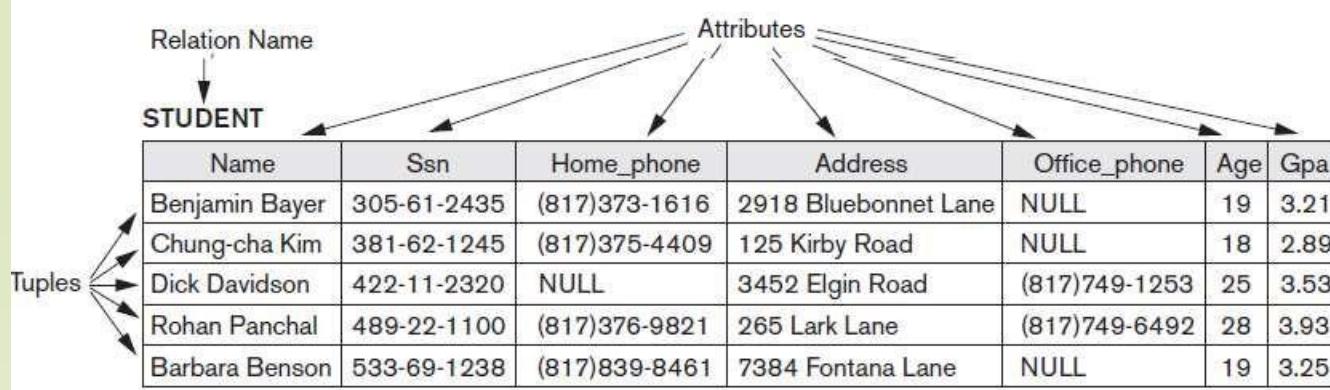
- No two tuples can have the same combination of values for all their attributes.
- There are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes
- Suppose that we denote 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:

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

- such set of attributes SK is called a **superkey** of the relation schema R

- **superkey**
 - specifies a uniqueness constraint that no two distinct tuples in any state r of R can have the same value for SK
- **Key**
 - Superkey of R
 - Removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more
 - satisfies two properties:
 1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key
 2. It is a minimal superkey—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold

- Example: Consider the STUDENT relation



- The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn
- Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey
- 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

- **Candidate key**

- 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 has two candidate keys:
License_number and Engine_serial_number

CAR

<u>License_number</u>	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

- Primary key of the relation
 - Designated among candidate keys
 - Underline attribute
- Other candidate keys are designated as **unique keys** and are not underlined
- Another constraint on attributes specifies whether NULL values are or are not permitted.
- For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

Relational Databases and Relational Database Schemas

Relational database schema S

- Set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$
- Set of integrity constraints IC

■ Relational database state

- Set of relation states $DB = \{r_1, r_2, \dots, r_m\}$
- Each r_i is a state of R and such that the r_i relation states satisfy integrity constraints specified in IC

■ Example: relational database schema

COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS,
PROJECT, WORKS_ON, DEPENDENT}

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	gender	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>	<u>Dependent_name</u>	gender	Bdate	Relationship
-------------	-----------------------	--------	-------	--------------

Figure : Schema diagram for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	gender	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

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

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	gender	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

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

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

- **Invalid state**
 - Does not obey all the integrity constraints
- **Valid state**
 - Satisfies all the constraints in the defined set of integrity constraints IC

- Integrity constraints are specified on a database schema and are expected to hold on every valid database state of that schema.
- In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational

model:

- entity integrity and
- referential integrity.

Integrity, Referential Integrity and Foreign Keys

Entity integrity constraint

- states that no primary key value can be NULL

▪ Referential integrity constraint

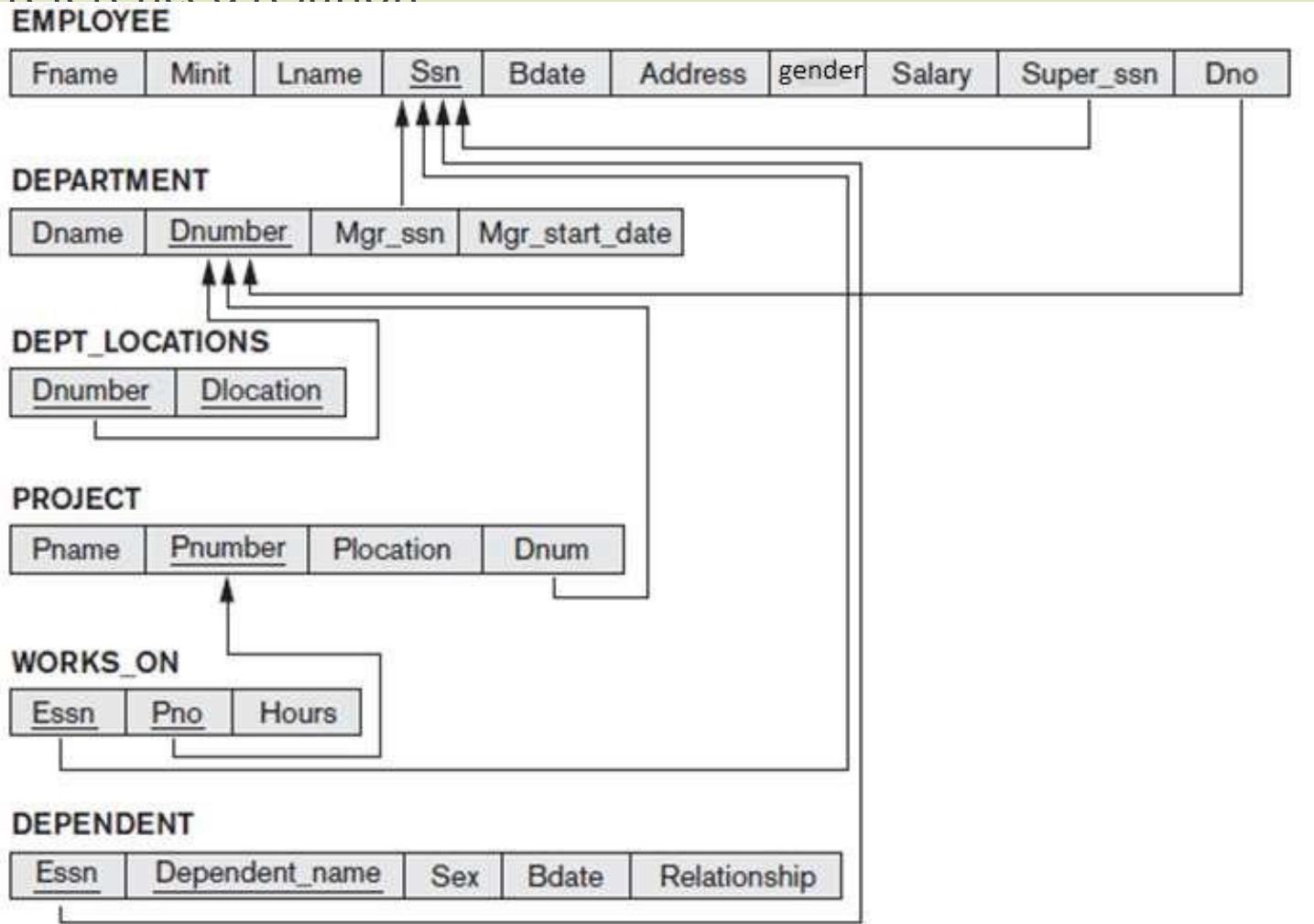
- Specified between two relations
- Maintains consistency among tuples in two relations
- Informally - a tuple in one relation that refers to another relation must refer to an existing tuple in that relation

- **Foreign key**

- 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:
 1. 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
 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 $t_1[\text{FK}] = t_2[\text{PK}]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2

- In the definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation**.
- 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
 - 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



Other Types of Constraints

- **Semantic integrity constraints**

- May have to be specified and enforced on a relational database
- Use **triggers** and **assertions**
- More common to check for these types of constraints within the application programs

- **Functional dependency constraint**

- Establishes a functional relationship among two sets of attributes X and Y
- Value of X determines a unique value of Y

- **State constraints(static constraints)**

- Define the constraints that a valid state of the database must satisfy

- **Transition constraints(dynamic constraints)**

Update Operations, Transactions, and Dealing with Constraint Violations

- The operations of the relational model can be categorized into **retrievals and updates**
- There are three basic operations that can change the states of relations in the database:
 1. Insert - one or more new tuples in a relation
 2. Delete- delete tuples
 3. Update (or Modify)-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.

The Insert Operation

provides a list of attribute values for a new tuple t that is to be inserted into a relation R

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

Operation:

Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4>

Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected

Operation:

Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4>

Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

Operation:

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357
Windswept, Katy, TX', F, 28000, '987654321', 7>

Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

Operation:

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357
Windy Lane,Katy, TX', F, 28000, NULL, 4>

Result: This insertion satisfies all constraints, so it is acceptable.

- If an insertion violates one or more constraints, the default option is to reject the insertion
- it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected
- An attempt to correct the reason for rejecting the insertion

The Delete Operation

- violate only referential integrity
- occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database

Operation:

Delete the WORKS_ON tuple with Essn = '999887777' and Pno =10.

Result: This deletion is acceptable and deletes exactly one tuple.

Operation:

Delete the EMPLOYEE tuple with Ssn = '999887777'.

Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

Operation:

Delete the EMPLOYEE tuple with Ssn = '333445555'

Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

- Several options are available if a deletion operation causes a violation
 - 1. **restrict** - is to reject the deletion
 - 2. **cascade**, is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted
 - 3. **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

The Update Operation

- used to change the values of one or more attributes in a tuple (or tuples) of some relation R
- specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified

Operation:

Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.

Result: Acceptable.

Operation:

Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.

Result: Unacceptable, because it violates referential integrity.

Operation:

Update the Ssn of the EMPLOYEE tuple with Ssn =
'999887777' to '987654321'.

Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn

The Transaction Concept

- | A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database
- | At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema
- | A single transaction may involve any number of retrieval operations and any number of update operations.
- | These retrievals and updates will together form an atomic unit of work against the database
- | For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.

Relational Algebra

- Relational algebra is the basic set of operations for the relational model
- These operations enable a user to specify basic retrieval requests as relational algebra expressions.
- The result of an operation is a new relation, which may have been formed from one or more input relations

- The relational algebra is very important for several reasons
 - provides a formal foundation for relational model operations.
 - used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs)
 - some of its concepts are incorporated into the SQL standard query language for RDBMSs

Unary Relational Operations: SELECT and PROJECT

The SELECT Operation

- The SELECT operation denoted by σ (sigma) is used to select a subset of the tuples from a relation based on a **selection condition**
- The selection condition acts as a **filter**
- Keeps only those tuples that satisfy the qualifying condition
- Tuples satisfying the condition are selected whereas the other tuples are discarded (filtered out)
- The SELECT operation can also be visualized as a horizontal partition of the relation into two sets of tuples
 - those tuples that satisfy the condition and are selected
 - those tuples that do not satisfy the condition and are discarded

- In general, the select operation is denoted by

$$\sigma_{<\text{selection condition}>}(R)$$

where

- the symbol σ is used to denote the select operator
- the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
- tuples that make the condition **true** are selected
 - appear in the result of the operation
- tuples that make the condition **false** are filtered out
 - discarded from the result of the operation

- The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>

or

<attribute name> <comparison op> <attribute name>

where

<attribute name> is the name of an attribute of R ,

<comparison op> is one of the operators $\{=, <, \leq, >, \geq, \neq\}$, and

<constant value> is a constant value from the attribute domain.

- Clauses can be connected by the standard Boolean operators **and**, **or**, and **not** to form a general selection condition

I Examples:

- Select the EMPLOYEE tuples whose department number is 4:

$$\sigma_{DNO=4} (\text{EMPLOYEE})$$

- Select the employee tuples whose salary is greater than \$30,000:

$$\sigma_{\text{SALARY} > 30,000} (\text{EMPLOYEE})$$

- ## I Select the tuples for all employees who either work in department 4 and make over \$25,000 per year, or work in department 5 and make over \$30,000

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

- The result of a SELECT operation can be determined as follows:
 - The <selection condition> is applied independently to each individual tuple t in R
 - If the condition evaluates to TRUE, then tuple t is selected
 - All the selected tuples appear in the result of the SELECT operation
 - The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:
 - (cond1 AND cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
 - (cond1 OR cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
 - (NOT cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

- The SELECT operator is **unary**; that is, it is applied to a single relation
- The **degree** of the relation resulting from a SELECT operation is the same as the degree of R
- The number of tuples in the resulting relation is always less than or equal to the number of tuples in R. That is,
$$|\sigma_c(R)| \leq |R| \text{ for any condition } C$$
- The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

- The SELECT operation is **commutative**; that is,

$$\sigma_{\langle \text{cond}_1 \rangle}(\sigma_{\langle \text{cond}_2 \rangle}(R)) = \sigma_{\langle \text{cond}_2 \rangle}(\sigma_{\langle \text{cond}_1 \rangle}(R))$$

Hence, a sequence of SELECTs can be applied in any order

- we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{\langle \text{cond}_1 \rangle}(\sigma_{\langle \text{cond}_2 \rangle}(\dots(\sigma_{\langle \text{cond}_n \rangle}(R)) \dots)) = \sigma_{\langle \text{cond}_1 \rangle} \text{ AND}_{\langle \text{cond}_2 \rangle} \text{ AND} \dots \text{ AND}_{\langle \text{cond}_n \rangle}(R)$$

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

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

would map to the following SQL query:

SELECT * FROM EMPLOYEE

WHERE Dno=4 AND Salary>25000;

The **PROJECT** Operation

The **PROJECT** operation selects certain columns from the table and discards the other columns

- Used when we are interested in only certain attributes of a relation
- The result of the PROJECT operation can be visualized as a vertical partition of the relation into two relations:
 - one has the needed columns (attributes) and contains the result of the operation
 - the other contains the discarded columns.

- The general form of the PROJECT operation is

$$\pi_{\langle \text{attributelist} \rangle}(R)$$

where

π (pi) - symbol used to represent the PROJECT operation,

$\langle \text{attributelist} \rangle$ - desired sublist of attributes from the attributes of relation R.

- The result of the PROJECT operation has only the attributes specified in $\langle \text{attribute list} \rangle$ in the same order as they appear in the list
- Hence, its **degree** is equal to the number of attributes in $\langle \text{attribute list} \rangle$

- Example : to list each employee's first and last name and salary we can use the PROJECT operation as follows:

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

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

- If the attribute list includes only nonkey attributes of R, duplicate tuples are likely to occur
- The result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**
- For example, consider the following PROJECT operation:

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



gender	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

- The number of operation is always less than or equal to the number of tuples in R

lation resulting from a PROJECT or equal to the number of tuples

- Commutativity does not hold on PROJECT

$$\pi_{\langle \text{list1} \rangle} (\pi_{\langle \text{list2} \rangle} (R)) = \pi_{\langle \text{list1} \rangle} (R)$$

as long as $\langle \text{list2} \rangle$ contains the attributes in $\langle \text{list1} \rangle$; otherwise, the left-hand side is an incorrect expression.

- In SQL, the PROJECT attribute list is specified in the SELECT clause of a query
- For example, the following operation: $\pi_{\text{gender}, \text{Salary}}(\text{EMPLOYEE})$ would correspond to the following SQL query:

SELECT DISTINCT gender, Salary

FROM EMPLOYEE

Sequences of Operations and the RENAME Operation

For most queries, we need to apply several relational algebra operations one after the other

- Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations
- In the latter case, we must give names to the relations that hold the intermediate results
- For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation

- We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

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

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

- Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$\text{DEP5_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname}, \text{Lname}, \text{Salary}}(\text{DEP5_EMPS})$$

- We can also use this technique to **rename** the attributes in the intermediate and result relations
- To rename the attributes in a relation, we simply list the new attribute names in parentheses

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

$R(\text{First_name}, \text{Last_name}, \text{Salary}) \leftarrow \pi_{\text{Fname}, \text{Lname}, \text{Salary}}(\text{TEMP})$

TEMP

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
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

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

- If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order
- For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list
- We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator.

- The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

1. $\rho_{S(B_1, B_2, \dots, B_n)}(R)$ ρ (rho) – RENAME operator
2. $\rho S(R)$ S – new relation name
3. $\rho_{(B_1, B_2, \dots, B_n)}(R)$ B_1, B_2, \dots, B_n - new attribute names

- The first expression renames both the relation and its attributes
- The second renames the relation only and
- The third renames the attributes only
- If the attributes of R are (A_1, A_2, \dots, A_n) in that order, then each A_i is renamed as B_i .

- Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

```
SELECT E.Fname AS First_name,
```

```
E.Lname AS Last_name,
```

```
E.Salary AS Salary
```

```
FROM EMPLOYEE AS E
```

```
WHERE E.Dno=5,
```

Relational Algebra Operations from Set Theory

The UNION, INTERSECTION, and MINUS Operations

- **UNION:** The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S . Duplicate tuples are eliminated.
- **INTERSECTION:** The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S .
- **SET DIFFERENCE (or MINUS):** The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S .

STUDENT

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

INSTRUCTOR

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

STUDENT \cup INSTRUCTOR.

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

STUDENT

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

INSTRUCTOR

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

STUDENT \cap INSTRUCTOR.

Fn	Ln
Susan	Yao
Ramesh	Shah



STUDENT

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

INSTRUCTOR

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

STUDENT – INSTRUCTOR.

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

INSTRUCTOR – STUDENT.

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

Example: to retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5

$DEP5_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$

$RESULT1 \leftarrow \pi_{Ssn}(DEP5_EMPS)$

$RESULT2(Ssn) \leftarrow \pi_{Super_ssn}(DEP5_EMPS)$

$RESULT \leftarrow RESULT1 \cup RESULT2$

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	gender	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

Single relational algebra expression,:

Result $\leftarrow \Pi_{Ssn} (\sigma_{Dno=5} (\text{EMPLOYEE})) \cup$

$\Pi_{\text{Super_ssn}} (\sigma_{Dno=5} (\text{EMPLOYEE}))$

- These are **binary** operations; that is, each is applied to two sets (of tuples)
- When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called **union compatibility** or **type 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 compatible** (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 number of attributes and each corresponding pair of attributes has the same domain.

- Both UNION and INTERSECTION are *commutative operations*; that is,
$$R \cup S = S \cup R \text{ and } R \cap S = S \cap R$$
- Both UNION and INTERSECTION can be treated as n -ary operations applicable to any number of relations because both are also *associative operations*; that is,

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

- The MINUS operation is *not commutative*; that is, in general,
$$R - S \neq S - R$$
- INTERSECTION can be expressed in terms of union and set difference as follows:

$$R \cap S = ((R \cup S) - (R - S)) - (S - R)$$

- In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN denoted by \times is a binary set operation, but the relations on which it is applied do *not* have to be union compatible.

- This set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set)
- In general, the result of $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, in that order.

- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S .
- Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples
- Example: suppose that we want to retrieve a list of names of each female employee's dependents.

$\text{FEMALE_EMPS} \leftarrow \sigma_{\text{gender}='F'}(\text{EMPLOYEE})$

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

$\text{EMP_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$

$\text{ACTUAL_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP_DEPENDENTS})$

$\text{RESULT} \leftarrow \pi_{\text{Fname}, \text{Lname}, \text{Dependent_name}} (\text{ACTUAL_DEPENDENTS})$

FEMALE_EMPS

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

EMPNAMES

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

EMP_DEPENDENTS

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

ACTUAL_DEPENDENTS

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

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

- The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations.
- We can SELECT related tuples only from the two relations by specifying an appropriate selection condition after the Cartesian product
- In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables

Binary Relational Operations: JOIN and DIVISION

The JOIN Operation

- The JOIN operation, denoted by \bowtie is used to combine related tuples from two relations into single “longer” tuples.
- it allows us to process relationships among relations
- 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

$$R \quad \bowtie \quad S$$

<join condition>

- Example: retrieve the name of the manager of each department
- To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple

$$\begin{aligned} \text{DEPT_MGR} &\leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE} \\ \text{RESULT} &\leftarrow \pi_{\text{Dname}, \text{Lname}, \text{Fname}}(\text{DEPT_MGR}) \end{aligned}$$

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

- The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ in that order;
- Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition. This is the main difference between CARTESIAN PRODUCT and JOIN.
- In JOIN, only combinations of tuples satisfying the join condition appear in the result, whereas in the CARTESIAN PRODUCT all combinations of tuples are included in the result.
- The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples.

- Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q as a single combined tuple.
 - A general join condition is of the form
$$<\text{condition}> \text{ AND } <\text{condition}> \text{ AND...AND } <\text{condition}>$$
where each $<\text{condition}>$ is of the form $A_i \theta B_j$, A_i is an attribute of R , B_j is an attribute of S , A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$.
- A JOIN operation with such a general join condition is called a **THETA JOIN**
- Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result.

Variations of JOIN: The EQUIJOIN and NATURAL JOIN

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

- In the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple.
- For example the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes requires the values to be identical in every tuple in the result.

- The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.
- Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.
- first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

$\text{PROJ_DEPT} \leftarrow \text{PROJECT} * \rho_{(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})} (\text{DEPARTMENT})$

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

$\text{DEPT} \leftarrow \rho_{(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})}(\text{DEPARTMENT})$

$\text{PROJ_DEPT} \leftarrow \text{PROJECT} * \text{DEPT}$

- The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations.

PROJ_DEPT

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

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

DEPT_LOCS \leftarrow DEPARTMENT * DEPT_LOCATIONS

DEPT_LOCS

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

by equating each pair of join attributes that have the same name in the two relations and combining these conditions with **AND**.

- if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples.

- A more general, but nonstandard definition for NATURAL JOIN
is

$$Q \leftarrow R *_{(\langle \text{list1} \rangle), (\langle \text{list2} \rangle)} S$$

$\langle \text{list1} \rangle$: list of i attributes from R ,

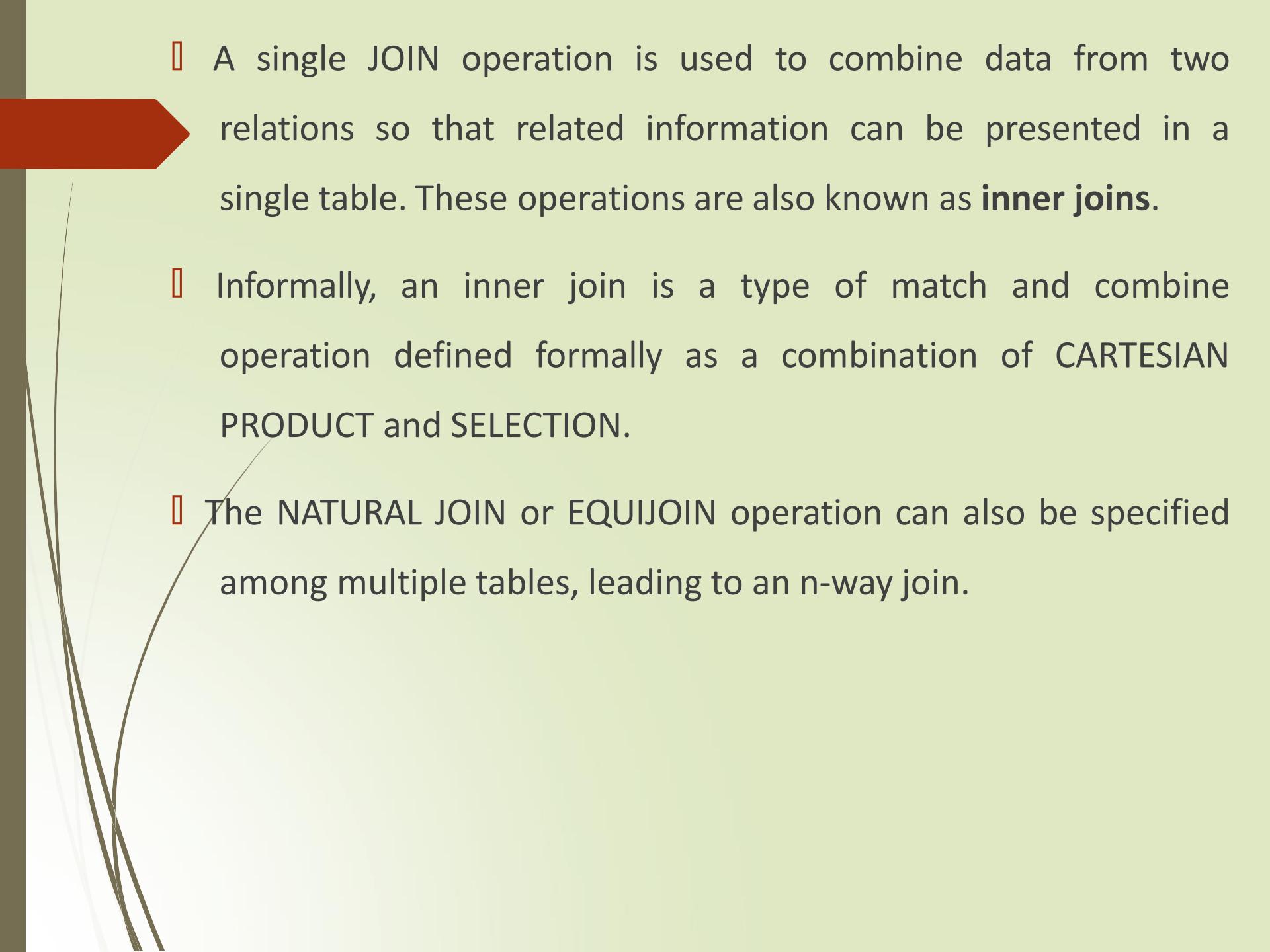
$\langle \text{list2} \rangle$: list of i attributes from S

The lists are used to form equality comparison conditions
between pairs of corresponding attributes

The conditions are then ANDed together

Only the list corresponding to attributes of the first relation
 $R - \langle \text{list1} \rangle -$ is kept in the result Q .

- In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \Join_{\text{condition}} S$ will have between zero and $n_R * n_S$ tuples.
- The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity**, which is a property of each join condition.
- If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

- 
- | A single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**.
 - | Informally, an inner join is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION.
 - | The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an n-way join.

- | For example, consider the following three-way join:


$$((\text{PROJECT} \bowtie_{Dnum=Dnumber} \text{DEPARTMENT}) \bowtie_{Mgr_ssn=Ssn} \text{EMPLOYEE})$$

- | This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager.
- | The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

- 
- || In SQL, JOIN can be realized in several different ways.
 - The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions.
 - The second way is to use a nested relation
 - Another way is to use the concept of joined tables

A Complete Set of Relational Algebra Operations

- | The set of relational algebra operations $\{\sigma, \pi, U, \rho, -, \times\}$ is a **complete** set; that is, any of the other original relational algebra operations can be expressed as a sequence of operations from this set.
- | For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

- | As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation,

$$R \bowtie_{<\text{condition}>} S \equiv \sigma_{<\text{condition}>} (R \times S)$$

- 
- || Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.
 - || Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra.

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{<\text{join condition}>} R_2$, OR $R_1 \bowtie_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{<\text{join condition}>} R_2$, OR $R_1 *_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2 ; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2 .	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

The DIVISION Operation

- | The DIVISION operation, denoted by \div , is useful for a special kind of query that sometimes occurs in database applications.
- | An example is Retrieve the names of employees who work on all the projects that 'John Smith' works on.
- | To express this query using the DIVISION operation, proceed as follows.
 - First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

```
SMITH ←  $\sigma_{Fname='John' \text{ AND } Lname='Smith'}(EMPLOYEE)$ 
SMITH_PNOS ←  $\pi_{Pno}(WORKS\_ON \bowtie_{Esn=Ssn} SMITH)$ 
```

- 
- 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:

$$\text{SSN_PNOS} \leftarrow \pi_{\text{Essn}, \text{Pno}}(\text{WORKS_ON})$$

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

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

(a)

SSN_PNOS

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

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

- | In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where the attributes of R are a subset of the attributes of S ; that is, $X \subseteq Z$.
- | Let Y be the set of attributes of R that are not attributes of S ; that is, $Y = Z - X$ (and hence $Z = X \cup Y$).
- | The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ for every tuple t_S in S . This means that, for a tuple t to appear in the result T of t

- Figure below illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$.

The diagram illustrates the division operation with three relations:

- R**: A relation with columns **A** and **B**. It contains 12 tuples: (a1, b1), (a2, b1), (a3, b1), (a4, b1), (a1, b2), (a3, b2), (a2, b3), (a3, b3), (a4, b3), (a1, b4), (a2, b4), and (a3, b4).
- S**: A relation with column **A**. It contains 4 tuples: a1, a2, a3, and a4.
- T**: A relation with column **B**. It contains 4 tuples: b1, b2, b3, and b4.

- the tuples (values) $b1$ and $b4$ appear in R in combination with all three tuples in S ; that is why they appear in the resulting relation T .
- All other values of B in R do not appear with all the tuples in S and are not selected: $b2$ does not appear with $a2$, and $b3$ does not appear with $a1$.

- 
- | The DIVISION operation can be expressed as a sequence of π , \times , and $-$ operations as follows:

$$\begin{aligned}T1 &\leftarrow \pi_Y(R) \\T2 &\leftarrow \pi_Y((S \times T1) - R) \\T &\leftarrow T1 - T2\end{aligned}$$

Exercise

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

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

Notation for Query Trees

- Query tree (query evaluation tree or query execution tree) is used in relational systems to represent queries internally
- A **query tree** is a tree data structure that corresponds to a relational algebra expression.
- It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes.

- An execution of the query tree consists of executing an internal node operation whenever its operands represented by its child nodes are available, and
- then replacing that internal node by the relation that results from executing the operation.
- The execution terminates when the root node is executed and produces the result relation for the query.
- Example: 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.

$$\pi_{Pnumber, Dnum, Lname, Address, Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr_ssn=Ssn}(EMPLOYEE))$$

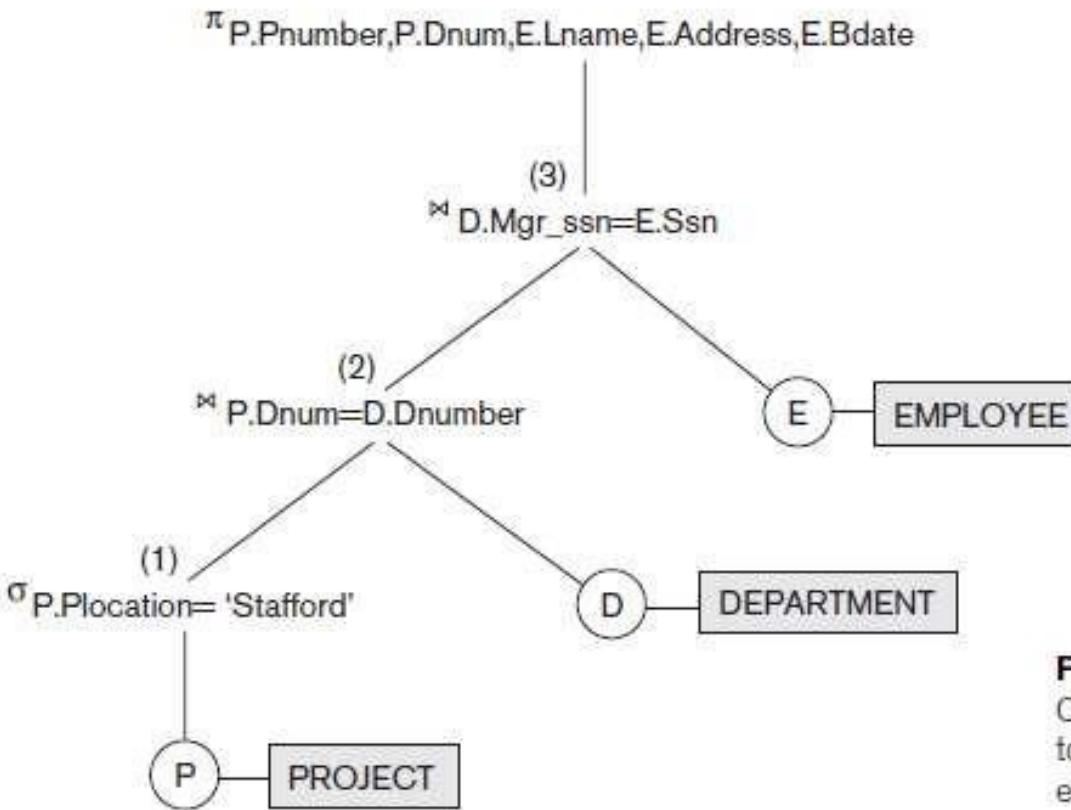
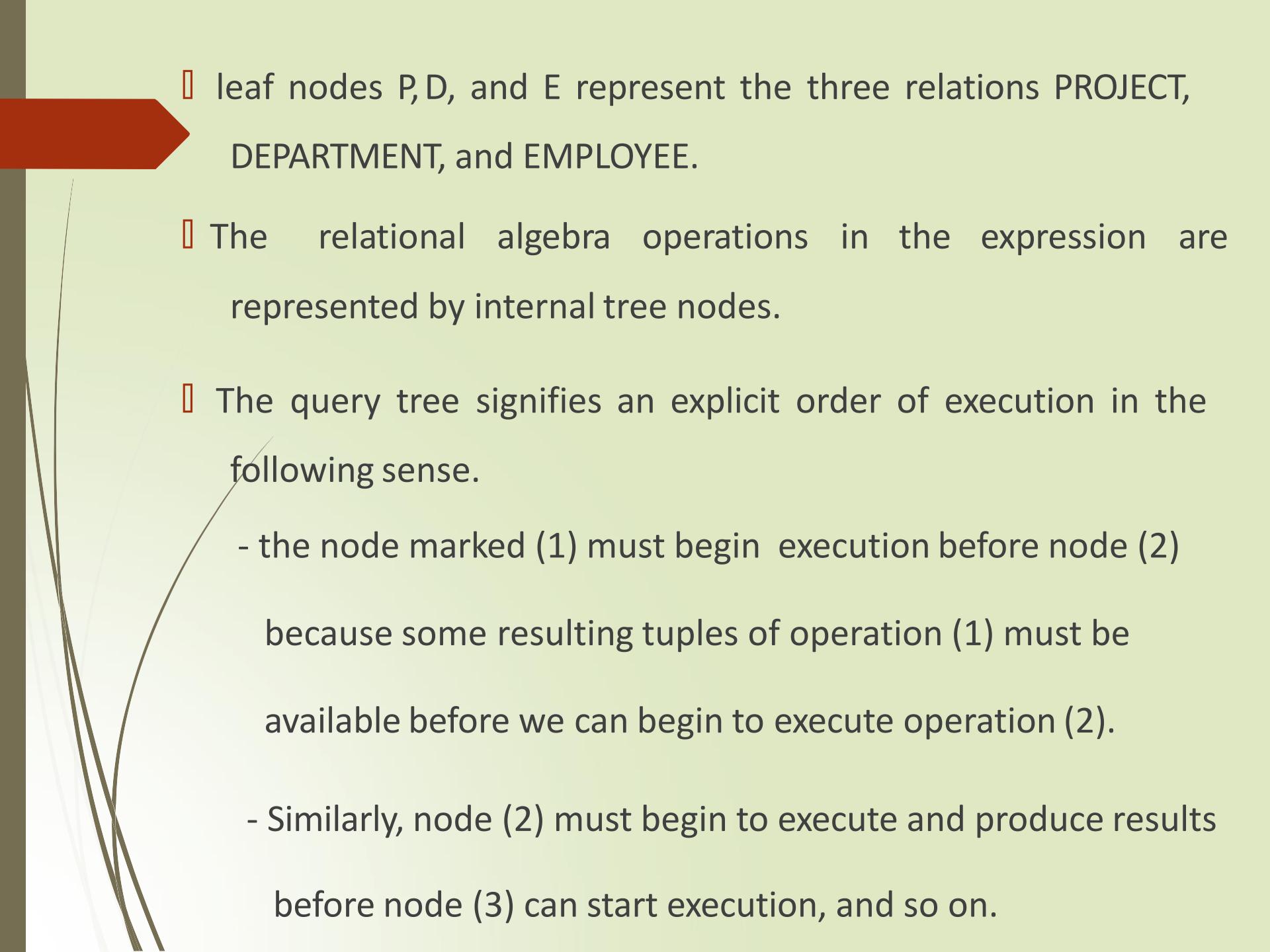


Figure 6.9
 Query tree corresponding
 to the relational algebra
 expression for Q2.

- 
- | leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE.
 - | The relational algebra operations in the expression are represented by internal tree nodes.
 - | The query tree signifies an explicit order of execution in the following sense.
 - the node marked (1) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2).
 - Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on.

- a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra.

Additional Relational Operations

operations enhance the expressive power of the original relational algebra

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:

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

where F_1, F_2, \dots, F_n are functions over the attributes in relation R and may involve arithmetic operations and constant values.

- helpful when developing reports where computed values have to be produced in the columns of a query result

- 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.

- generalized projection combined with renaming :

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

$\text{Years_service}, 0.25 * \text{Salary}}(\text{EMPLOYEE}).$

Aggregate Functions and Grouping

- used in simple statistical queries that summarize information from the database tuples
- Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM
- The COUNT function is used for counting tuples or values.
- Examples : retrieving the average or total salary of all employees or the total number of employee tuples
- grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group.
- Example : group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department. We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.

■ AGGREGATE FUNCTION operation can be defined by using the symbol Σ :

$$\langle \text{grouping attributes} \rangle \Sigma \langle \text{function list} \rangle (R)$$

Where

$\langle \text{grouping attributes} \rangle$: list of attributes of the relation specified in R

$\langle \text{function list} \rangle$: list of ($\langle \text{function} \rangle \langle \text{attribute} \rangle$) pairs.

$\langle \text{function} \rangle$ - such as SUM, AVERAGE, MAXIMUM,
MINIMUM,COUNT

$\langle \text{attribute} \rangle$ is an attribute of the relation specified by R

■ The resulting relation has the grouping attributes plus one attribute for each element in the function list

Example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes

$\rho_R(Dno, No_of_employees, Average_sal)(Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE))$

The aggregate function operation.

- a. $\rho_R(Dno, No_of_employees, Average_sal)(Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE))$.
- b. $Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE)$.
- c. $\Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE)$.

R

(a)	Dno	No_of_employees	Average_sal
	5	4	33250
	4	3	31000
	1	1	55000

(b)

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

(c)

Count_ssn	Average_salary
8	35125

Recursive Closure Operations

- operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor
- Example : retrieve all supervisees of an employee e at all levels—that is, all employees e' directly supervised by e , all employees e'' directly supervised by each employee e' , all employees e''' directly supervised by each employee e'' and so on.

```
BORG_SSN ← πSsn(σFname='James' AND Lname='Borg(EMPLOYEE))
SUPERVISION(Ssn1, Ssn2) ← πSen,Super_ssn(EMPLOYEE)
RESULT1(Ssn) ← πSsn1(SUPERVISION ⋈Sen2=Ssn BORG_SSN)
```

SUPERVISION

(Borg's Ssn is 888665555)

(Ssn) (Super_ssn)

Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

- To retrieve all employees supervised by Borg at level 2—that is, all employees e'' supervised by some employee e' who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

$$\text{RESULT2}(\text{Ssn}) \leftarrow \pi_{\text{Ssn}_1}(\text{SUPERVISION} \bowtie_{\text{Ssn}_2=\text{Ssn}} \text{RESULT1})$$

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by
Borg's subordinates)

- To get both sets of employees supervised at levels 1 and 2 by 'James Borg', we can apply the UNION operation to the two results, as follows:

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

OUTER JOIN Operations

- The JOIN operations match tuples that satisfy the join condition.
- For example, for a NATURAL JOIN operation $R * S$, only tuples from R that have matching tuples in S—and vice versa—appear in the result.
- Hence, tuples without a matching (or related) tuple are eliminated from the JOIN result.
 - Tuples with NULL values in the join attributes are also eliminated.
 - This type of join, where tuples with no match are eliminated, is known as an **inner join**.

- A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation.
- For example, suppose that we want a list of all employee names as well as the name of the departments they manage if they happen to manage a department; if they do not manage one, we can indicate it with a NULL value.
- We can apply an operation **LEFT OUTER JOIN**, denoted by  , to retrieve the result as follows:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} \bowtie_{\text{Ssn}=\text{Mgr_ssn}} \text{DEPARTMENT})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname}, \text{Minit}, \text{Lname}, \text{Dname}}(\text{TEMP})$$

- 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.

RESULT

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

- A similar operation, **RIGHT OUTER JOIN**, denoted by 

keeps every tuple in the *second*, or right, relation S in the result of $R \bowtie S$.

- A third operation, **FULL OUTER JOIN**, denoted by , keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

The OUTER UNION Operation

- developed to take the union of tuples from two relations that have some common attributes, but are not union (type) compatible.
- This operation will take the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible**, meaning that only some of their attributes, say X , are union compatible
- The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation $T(X, Y, Z)$.

- 
- Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X] = t_2[X]$. These will be combined (unioned) into a single tuple in t .
 - Tuples in either relation that have no matching tuple in the other relation are padded with NULL values.
 - For example, an OUTER UNION can be applied to two relations whose schemas are:
 - STUDENT(Name, Ssn, Department, Advisor)
 - INSTRUCTOR(Name, Ssn, Department, Rank)
 - Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department.

- █ All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result.
- █ Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute.
- █ A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes
- █ The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)

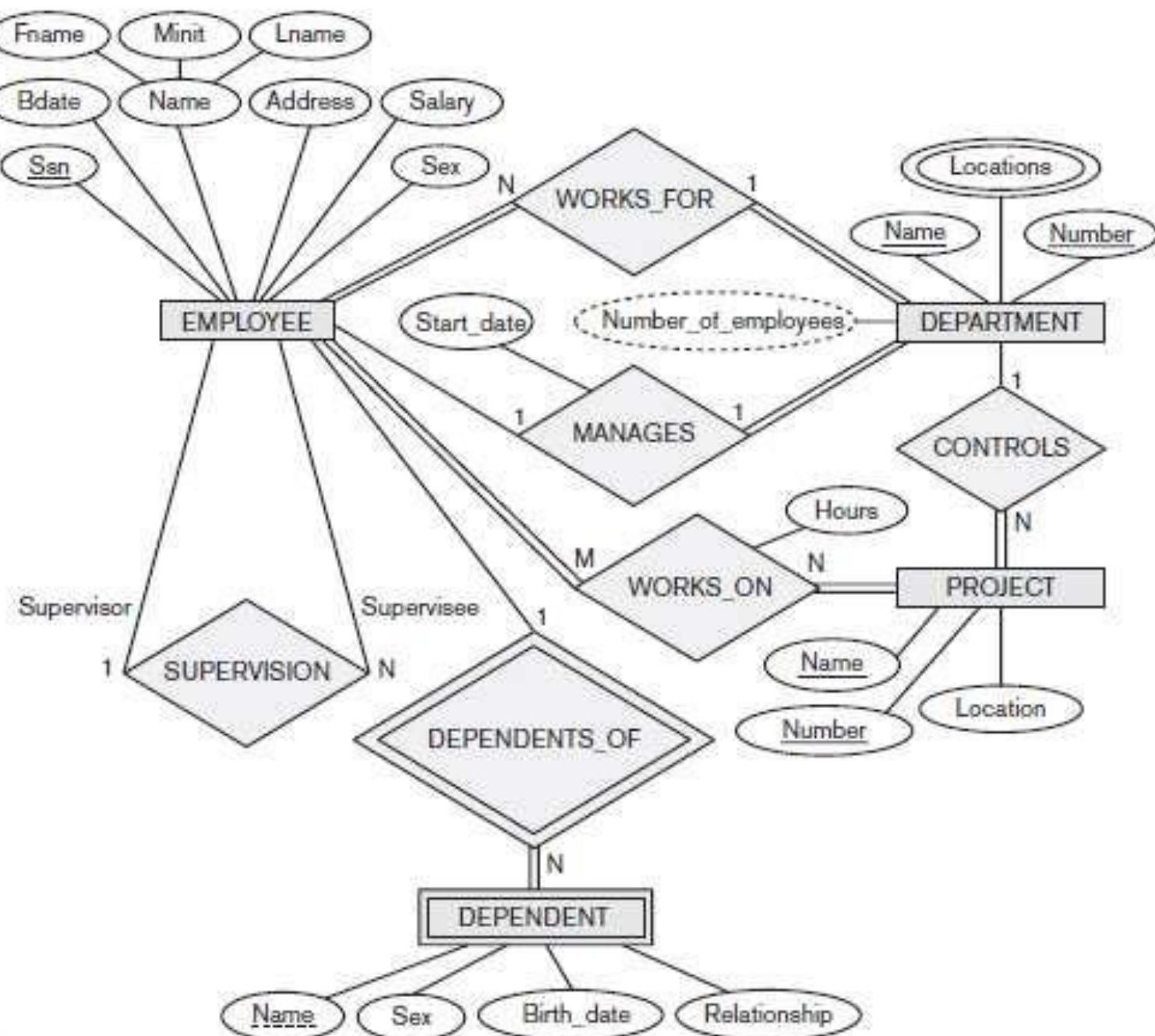
Self-Learning Topic

Examples of Queries in Relational Algebra

- There are 7 example queries out of which 2 queries are discussed in class

Relational Database Design using ER-to-Relational mapping

- procedures to create a relational schema from an Entity-Relationship (ER)
- convert the basic ER model constructs—entity types (weak) and binary relationships (with various structural constraints), n -ary relationships, and attributes (simple, composite, and multivalued)—into relations.



Step 1: Mapping of Regular Entity Types.

For each regular entity type, create a relation R that includes all the simple attributes of E

- Include only the simple component attributes of a composite attribute
- Choose one of the key attributes of E as the primary key for R
- If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R.
- If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R

- In our example-COMPANY database, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT
- we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively

- The relations that are created from the mapping of entity type **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary
-------	-------	-------	-----	-------	---------	-----	--------

DEPARTMENT

Dname	Dnumber
-------	---------

PROJECT

Pname	Pnumber	Plocation
-------	---------	-----------

Step 2: Mapping of Weak Entity Types

For each weak entity type, create a relation R and include all simple attributes of the entity type as attributes of R

- Include primary key attribute of owner as foreign key attributes of R
- In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT
- We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it as Essn
- The primary key of the DEPENDENT relation is the combination {Essn,Dependent_name}, because Dependent_name is the partial key of DEPENDENT

- It is common to choose the propagate (CASCADE) option for the referential triggered action on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity.
- This can be used for both ON UPDATE and ON DELETE.

DEPENDENT

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

Step 3: Mapping of Binary 1:1 Relationship Types

For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R

- There are three possible approaches:
 - the foreign key approach
 - the merged relationship approach
 - the crossreference or relationship relation approach

1. The foreign key approach

- Choose one of the relations— S , say—and include as a foreign key in S the primary key of T .

- Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S .
- In our example, we map the 1:1 relationship type 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 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

2. Merged relation approach:

merge the two entity types and the relationship into a single relation

- This is possible when *both participations are total*, as this would indicate that the two tables will have the exact same number of tuples at all times.

3. Cross-reference or relationship relation approach:

- 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.
- 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 an extra join operation when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types

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
- Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S
- In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION

- For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno.
- For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn.
- The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation.

Step 5: Mapping of Binary M:N Relationship Types

- For each binary M:N relationship type
 - Create a new relation S
 - Include primary key of participating entity types as foreign key attributes in S
 - Include any simple attributes of M:N relationship type
- In our example, we map the M:N relationship type WORKS_ON by creating the relation WORKS_ON
- We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively.

- We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type.
- The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}.

WORKS_ON		
<u>Essn</u>	<u>Pno</u>	Hours

- The propagate (CASCADE) option for the referential triggered action should be specified on the foreign keys in the relation corresponding to the relationship R, since each relationship instance has an existence dependency on each of the entities it relates.
- This can be used for both ON UPDATE and ON DELETE.

Step 6: Mapping of Multivalued Attributes

- For each multivalued attribute
 - Create a new relation
 - Primary key of R is the combination of A and K
 - If the multivalued attribute is composite, include its simple components
- In our example, we create a relation DEPT_LOCATIONS

- The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, while Dnumber—as foreign key—represents the primary key of the DEPARTMENT relation.
- The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}
- A separate tuple will exist in DEPT_LOCATIONS for each location that a department has
- The propagate (CASCADE) option for the referential triggered action should be specified on the foreign key in the relation R corresponding to the multivalued attribute for both ON UPDATE and ON DELETE.

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>	<u>Plocation</u>	Dnum
-------	----------------	------------------	------

WORKS_ON

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

DEPENDENT

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

Step 7: Mapping of *N*-ary Relationship Types

For each *n*-ary relationship type *R*

- Create a new relation *S* to represent *R*
- Include primary keys of participating entity types as foreign keys
- Include any simple attributes as attributes
- The primary key of *S* is usually a combination of all the foreign keys that reference the relations representing the participating entity types.
- For example, consider the relationship type SUPPLY
- This can be mapped to the relation SUPPLY whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

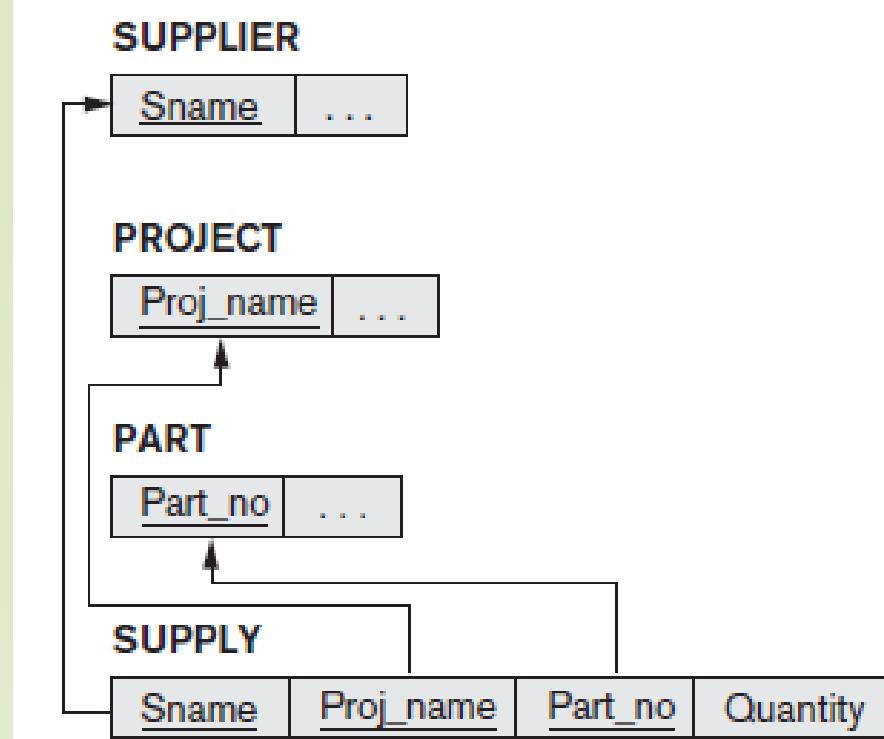
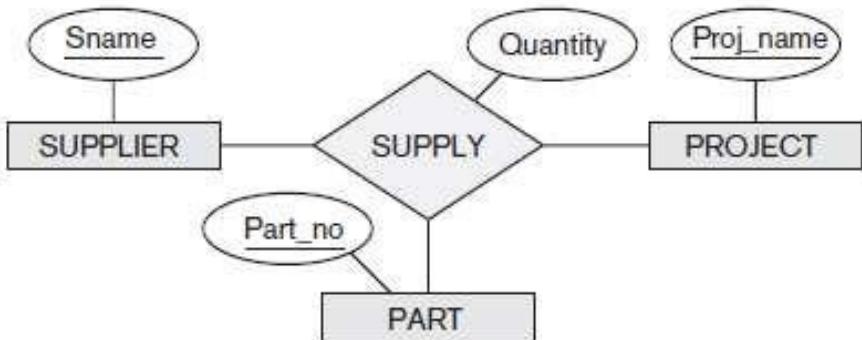


Table 9.1 Correspondence between ER and Relational Models

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 <i>two</i> 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

SQL: Structured Query Language

- SQL was called SEQUEL (Structured English Query Language) and was designed and implemented at IBM Research
- The SQL language may be considered one of the major reasons for the commercial success of relational databases
- It has statements for data definitions, queries, and updates.
- Facilities for defining views on the database,
- specifying security and authorization
- defining integrity constraints,
- specifying transaction controls
- rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++

SQL Data Definition and Data Types

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

Schema and Catalog Concepts in SQL

- An **SQL schema** is identified by a **schema name**, and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for each *element* in the schema.
- Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema.
- A schema is created via the CREATE SCHEMA statement
- For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'..

CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';

- In general, not all users are authorized to create schemas and schema elements.
- The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

- SQL uses the concept of a **catalog**—a named collection of **schemas** in an SQL environment.
- A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the schemas in the catalog and all the element descriptors in these schemas
- Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog
- Schemas within the same catalog can also share certain elements, such as domain definitions.

The CREATE TABLE Command in SQL

used to specify a new relation by giving it a name and specifying its attributes and initial constraints

- The attributes are specified first
 - each attribute is given a name
 - a data type to specify its domain of values and
 - any attribute constraints, such as NOT NULL
- The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command

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),  
FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),  
FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
```

Attribute Data Types and Domains in SQL

□ Basic data types

- **Numeric** data types
 - integer numbers of various sizes (**INTEGER** or **INT**, and **SMALLINT**)
 - 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 -precision, total number of decimal digits
 - j - scale, number of digits after the decimal point

- **Character-string** data types

fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters

- 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*
- 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

- 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.

- **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'
- 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.

- A 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.
- Only valid dates and times should be allowed by the SQL implementation.
- **TIME WITH TIME ZONE** data type includes an additional six positions for specifying the displacement from the standard universal time zone, which is in the range +13:00 to -12:59 in units of HOURS:MINUTES.
- If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

Additional data types

- **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier.
- **INTERVAL** data type. This specifies an **interval**—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp.
- Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

- It is possible to specify the data type of each attribute directly or a domain can be declared, and the domain name used with the attribute

Specification

- This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability.
- For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE AS CHAR(9);

- We can use SSN_TYPE in place of CHAR(9) for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn of DEPARTMENT, Essn of WORKS_ON, and Essn of DEPENDENT

Specifying Constraints in SQL

basic constraints that can be specified in SQL as part of table creation:

- key and referential integrity constraints
- Restrictions on attribute domains and NULLs
- constraints on individual tuples within a relation

Specifying Attribute Constraints and Attribute Defaults

- Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute
- This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL

- It is also possible to define a default value for an attribute by appending the clause **DEFAULT** <value> to an attribute definition
- The default value is included in any new tuple if an explicit value is not provided for that attribute

```
CREATE TABLE DEPARTMENT  
(...,  
    Mgr_ssn CHAR(9) NOT NULL DEFAULT  
'888665555',  
    -----  
    -----  
)
```

- Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition
- For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

Dnumber INT **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);

- The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement.
- For example, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER  
CHECK (D_NUM > 0 AND D_NUM < 21);
```
- We can then use the created domain D_NUM as the attribute type for all attributes that refer to department number such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

Specifying Key and Referential Integrity Constraints

The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation

- If a primary key has a single attribute, the clause can follow the attribute directly
- For example, the primary key of DEPARTMENT can be specified as:

Dnumber INT **PRIMARY KEY**;

The **UNIQUE** clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:

Dname VARCHAR(15) **UNIQUE**;

- Referential integrity is specified via the **FOREIGN KEY** clause

FOREIGN KEY (Super_ssn) **REFERENCES** EMPLOYEE(Ssn),

FOREIGN KEY (Dno) **REFERENCES** DEPARTMENT(Dnumber

- A referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified
- The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation, which is known as the ~~RESTRICT~~ option
- The schema designer can specify an alternative action to be taken by attaching a **referential triggered action** clause to any foreign key constraint
- The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE

- **FOREIGN KEY**(Dno) **REFERENCES** DEPARTMENT(Dnumber) **ON DELETE** SET DEFAULT **ON UPDATE** CASCADE
- **FOREIGN KEY** (Super_ssn) **REFERENCES** EMPLOYEE(Ssn) **ON DELETE** SET NULL **ON UPDATE** CASCADE
- **FOREIGN KEY** (Dnumber) **REFERENCES** DEPARTMENT(Dnumber) **ON DELETE** CASCADE **ON UPDATE** CASCADE
- In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT.

- The action for CASCADE ON DELETE is to delete all the referencing tuples
- whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples
- It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema.
- As a general rule, the CASCADE option is suitable for "relationship" relations such as WORKS_ON; for relations that represent multivalued attributes, such as

Giving Names to Constraints

- The names of all constraints within a particular schema must be unique
- A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint

Specifying Constraints on Tuples Using CHECK

- Other table constraints can be specified through additional CHECK clauses at the end of a CREATE TABLE statement
- These can be called **tuple-based** constraints because they apply to each tuple individually and are checked whenever a tuple is inserted or modified

- For example, suppose that the DEPARTMENT table had an additional attribute Dept_create_date, which stores the date when the department was created.
- Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date

CHECK(Dept_create_date <= Mgr_start_date);

Basic Retrieval Queries in SQL

The **SELECT-FROM-WHERE** Structure of Basic SQL Queries

SELECT <attribute list>

FROM <table list>

WHERE <condition>;

where

I <attribute list> is a list of attribute names whose values are

to

be retrieved by the query.

I <table list> is a list of the relation names required to

process

Query 1. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
SELECT Bdate, Address
```

```
FROM EMPLOYEE
```

```
WHERE Fname='John' AND Minit='B' AND Lname='Smith';
```

- The **SELECT** clause of SQL specifies the attributes whose values are to be retrieved, which are called the **projection attributes**
- The **WHERE** clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition**

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

SELECTFname, Lname, Address

FROM EMPLOYEE, DEPARTMENT

WHERE Dname='Research' **AND** Dnumber=Dno;

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

Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

- In SQL, the same name can be used for two or more attributes as long as the attributes are in different relations
- If this is the case, and a multitable query refers to two or more attributes with the same name, we must **qualify** the attribute name with the relation name to prevent ambiguity
- This is done by prefixing the relation name to the attribute name and separating the two by a period
- Example: Retrieve the name and address of all employees who work for the 'Research' department

```
SELECT Fname, EMPLOYEE.Name, Address  
FROM EMPLOYEE, DEPARTMENT  
WHERE DEPARTMENT.Name='Research' AND  
DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

- The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice
- **Query :** For each employee, retrieve the employee's first and last name and the first and last 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.Super_ssn=S.Ssn;
```

- In this case, we are required to declare alternative relation names E and S, called **aliases** or **tuple variables**, for the EMPLOYEE relation.
- An alias can follow the keyword **AS**, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S
- It is also possible to **rename** the relation attributes within the query in SQL by giving them aliases. For example, if we write
EMPLOYEE **AS** E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)
in the FROM clause, Fn becomes an alias for Fname, Mi for Minit,
Ln for Lname, and so on

Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result

- If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected
- **Queries 9 and 10.** Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

Q9: SELECT Ssn

FROM EMPLOYEE;

Q10: SELECT Ssn, Dname

FROM EMPLOYEE, DEPARTMENT;

- To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes
- For example, query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5

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

```
SELECT * FROM EMPLOYEE, DEPARTMENT  
WHERE Dname='Research' AND Dno=Dnumber;
```

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

Tables as Sets in SQL

- SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query.
- SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:
 - Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates
 - The user may want to see duplicate tuples in the result of a query.
 - When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

I 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.

Query : Retrieve the salary of every employee and all distinct salary values

SELECTA

SELECTDI

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

LOYEE;

- SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra
- There are
 - set union (**UNION**)
 - set difference (**EXCEPT**) and
 - set intersection (**INTERSECT**)
- The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result
- These set operations apply only to union-compatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations.

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

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

Substring Pattern Matching and Arithmetic Operators

several more features of SQL

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

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

```
SELECT Fname, Lname FROM EMPLOYEE WHERE Address  
LIKE '%Houston,TX%';
```

- To retrieve all employees who have r in the second position of name



```
SELECT Fname, Lname FROM EMPLOYEE WHERE fname  
LIKE '_r%';
```

- Another feature allows the use of arithmetic in queries.
- The standard arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains.

Query :Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

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

Query 14. 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) in Q14 is equivalent to the condition((Salary \geq 30000) **AND** (Salary \leq 40000)).

Ordering of Query Results

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

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

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

INSERT, DELETE, and UPDATE Statements in SQL

The **INSERT** Command

- **INSERT** is used to add a single tuple to a relation
- We must specify the relation name and a list of values for the tuple.
- The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command.
- **INSERT INTO EMPLOYEE VALUES** ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);
- **INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn)**
VALUES ('Richard', 'Marini', 4, '653298653');

The **DELETE** Command

- The **DELETE** command removes tuples from a relation.
- It includes a **WHERE** clause, similar to that used in an SQL query, to select the tuples to be deleted.
- Tuples are explicitly deleted from only one table at a time
- the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL

DELETE FROM EMPLOYEE WHERE Lname='Brown';

The **UPDATE** Command

- The **UPDATE** command is used to modify attribute values of one or more selected Tuples
- An additional **SET** clause in the **UPDATE** command specifies the attributes to be modified and their new values.
- For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use

UPDATE PROJECT **SET** Plocation = 'Bellaire', Dnum = 5

WHERE

Additional Features of SQL

- | various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification
- | SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases.
- | SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.
- | SQL has language constructs for specifying the *granting and revoking of privileges* to users.

- SQL has language constructs for creating triggers
 - SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational**.
- SQL and relational databases can interact with new technologies such as XML



MODULE 3

Advanced SQL

- 
- SQL Data Types and Schemas
 - Integrity Constraints
 - Authorization
 - Embedded SQL
 - Dynamic SQL
 - Functions and Procedural Constructs**
 - Recursive Queries**
 - Advanced SQL Features**

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and day
 - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
 - Example: **time** '09:00:30'
- **timestamp:** date plus time of day
 - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
 - Example: **interval** '1' day
 - Subtracting a date/time/timestamp value from another gives an interval value
 - Interval values can be added to date/time/timestamp values

Build-in Data Types in SQL (Cont.)

- | Can extract values of individual fields from date/time/timestamp
 - | Example: **extract (year from r.starttime)**
- | Can cast string types to date/time/timestamp
 - | Example: **cast <string-valued-expression> as date**
 - | Example: **cast <string-valued-expression> as time**

User-Defined Types

- I **create type** construct in SQL creates user-defined type

```
create type Dollars as numeric (12,2) final
```

- I **create domain** construct in SQL-92 creates user-defined domain types

```
create domain person_name char(20) not null
```

- I Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

Domain Constraints

- Domain constraints are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
 - Example: **create domain Dollars numeric(12, 2)**
create domain Pounds numeric(12,2)
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
 - However, we can convert type as below
(cast r.A as Pounds)
(Should also multiply by the dollar-to-pound conversion-rate)

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
 - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
 - **clob**: character large object -- object is a large collection of character data
 - When a query returns a large object, a pointer is returned rather than the large object itself.

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
 - A checking account must have a balance greater than \$10,000.00
 - A salary of a bank employee must be at least \$4.00 an hour
 - A customer must have a (non-null) phone number
 - An account balance cannot be null
 - No two accounts can have the same account number

Constraints on a Single Relation

- | **not null**
- | **primary key**
- | **unique**
- | **check (P)**, where P is a predicate

Not Null Constraint

- I Declare *branch_name* for *branch* is **not null**
***branch_name* char(15) not null**
- I Declare the domain *Dollars* to be **not null**
create domain Dollars numeric(12,2) not null
- I Any database modification that would cause a null to be inserted in an attribute declared to be not null generates an error diagnostic.

The Unique Constraint

- | **unique** (A_1, A_2, \dots, A_m)
- | The unique specification states that the attributes
 A_1, A_2, \dots, A_m
form a candidate key.
- | Candidate keys are permitted to be null (in contrast to primary
keys) unless they are explicitly declared to be not null.

The check clause

- **check** (P), where P is a predicate that must be satisfied by every tuple in a relation.

Example: Declare *branch_name* as the primary key for *branch* and ensure that the values of assets are non-negative.

```
create table branch
  (branch_name    char(15),
   branch_city     char(30),
   assets          integer,
   primary key (branch_name),
   check (assets >= 0))
```

The check clause (Cont.)

- | The **check** clause in SQL-92 permits domains to be restricted:
 - | Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.

```
create domain hourly_wage numeric(5,2)
constraint value_test check(value > = 4.00)
```
 - | The domain has a constraint that ensures that the hourly_wage is greater than 4.00
 - | The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

Referential Integrity

- | Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - | Example: If “Perryridge” is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch “Perryridge”.
- | Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
 - | The primary key clause lists attributes that comprise the primary key.
 - | The unique key clause lists attributes that comprise a candidate key.
 - | The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Referential Integrity in SQL – Example

```
create table customer  
(customer_name  char(20),  
customer_street  char(30),  
customer_city    char(30),  
primary key (customer_name ))
```

```
create table branch  
(branch_name      char(15),  
branch_city       char(30),  
assets            numeric(12,2),  
primary key (branch_name ),  
check (assets >= 0))
```

Referential Integrity in SQL – Example (Cont.)

```
create table account
(account_number  char(10),
branch_name    char(15),
balance        numeric(12, 2),
primary key (account_number),
foreign key (branch_name) references branch,
check (balance >= 0))

create table depositor
(customer_name char(20),
account_number char(10),
primary key (customer_name, account_number),
foreign key (account_number ) references account,
foreign key (customer_name ) references customer )
```

Assertions

- | An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- | An assertion in SQL takes the form
 - create assertion <assertion-name> check <predicate>**
- | When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
 - | This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- | Asserting
 - for all X , $P(X)$is achieved in a round-about fashion using
 - not exists X such that not $P(X)$

Assertion Example

- I Every loan has at least one borrower who maintains an account with a minimum balance of \$1000.00

```
create assertion balance_constraint check
  (not exists (
    select *
    from loan
    where not exists (
      select *
      from borrower, depositor, account
      where loan.loan_number = borrower.loan_number
        and borrower.customer_name = depositor.customer_name
        and depositor.account_number = account.account_number
        and account.balance >= 1000)))
```

Assertion Example

- I The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

create assertion *sum_constraint check*

```
(not exists (select *
             from branch
             where (select sum(amount )
                    from loan
                    where loan.branch_name =
                          branch.branch_name )
                   >= (select sum (amount )
                        from account
                        where loan.branch_name =
                          branch.branch_name )))
```

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- || The **grant** statement is used to confer authorization

grant <privilege list>

on <relation name or view name> **to** <user list>

- || <user list> is:

- || a user-id

- || **public**, which allows all valid users the privilege granted

- || A role (more on this in Chapter 8)

- || Granting a privilege on a view does not imply granting any privileges on the underlying relations.

- || The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- I **select:** allows read access to relation, or the ability to query using the view

- I Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *branch* relation:

- grant select on branch to** U_1, U_2, U_3

- I **insert:** the ability to insert tuples

- I **update:** the ability to update using the SQL update statement

- I **delete:** the ability to delete tuples.

- I **all privileges:** used as a short form for all the allowable privileges

- I more in Chapter 8

Revoking Authorization in SQL

- || The **revoke** statement is used to revoke authorization.

revoke <privilege list>

on <relation name or view name> **from** <user list>

- || Example:

revoke select on branch from U_1, U_2, U_3

- || <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- || If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- || If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- || All privileges that depend on the privilege being revoked are also revoked.

Embedded SQL

- | The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- | A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language constitute *embedded SQL*.
- | The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- | **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement> END_EXEC

Note: this varies by language (for example, the Java embedding uses

SQL { };)

Example Query

- From within a host language, find the names and cities of customers with more than the variable **amount** dollars in some account.
- Specify the query in SQL and declare a *cursor* for it

EXEC SQL

```
declare c cursor for
select customer_name, customer_city
from depositor, customer, account
where depositor.customer_name =customer.customer_name
and depositor.account_number=account.account_number
and account.balance > :amount
```

END-EXEC

Embedded SQL (Cont.)

- I The **open** statement causes the query to be evaluated

```
EXEC SQL open c END-EXEC
```

- I The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :cn, :cc END-EXEC
```

Repeated calls to **fetch** get successive tuples in the query result

- I A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- I The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c END-EXEC
```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
declare c cursor for
    select *
        from account
        where branch_name = 'Perryridge'
    for update
```

- To update tuple at the current location of cursor *c*

```
update account
    set balance = balance + 100
    where current of c
```

Dynamic SQL

- | Allows programs to construct and submit SQL queries at run time.
- | Example of the use of dynamic SQL from within a C program.

```
char* sqlprog = "update account  
    set balance =balance *1.05  
    where account_number =?"  
EXEC SQL prepare dynprog from :sqlprog;  
char account[10] ="A-101";  
EXEC SQL execute dynprog using :account;
```

- | The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.



ODBC and JDBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
 - Connect with the database server
 - Send SQL commands to the database server
 - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

ODBC

- | Open DataBase Connectivity(ODBC) standard
 - | standard for application program to communicate with a database server.
 - | application program interface (API) to
 - | open a connection with a database,
 - | send queries and updates,
 - | get back results.
- | Applications such as GUI, spreadsheets, etc. can use ODBC

ODBC (Cont.)

- | Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- | When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- | ODBC program first allocates an SQL environment, then a database connection handle.
- | Opens database connection using SQLConnect(). Parameters for SQLConnect:
 - | connection handle,
 - | the server to which to connect
 - | the user identifier,
 - | password
- | Must also specify types of arguments:
 - | SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
    {.... Do actual work ... }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

ODBC Code (Cont.)

- | Program sends SQL commands to the database by using SQLExecDirect
- | Result tuples are fetched using SQLFetch()
- | SQLBindCol() binds C language variables to attributes of the query result
 - | When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
 - | Arguments to SQLBindCol()
 - | ODBC stmt variable, attribute position in query result
 - | The type conversion from SQL to C .
 - | The address of the variable.
 - | For variable-length types like character arrays,
 - | The maximum length of the variable
 - | Location to store actual length when a tuple is fetched.
 - | Note: A negative value returned for the length field indicates null value
 - | Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

ODBC Code (Cont.)

I Main body of program

```
char branchname[80];
float balance;
int lenOut1, lenOut2;
HSTMT stmt;

SQLAllocStmt(conn, &stmt);
char *sqlquery ="select branch_name, sum (balance)
                  from account
                  group by branch_name";

SQLAlloc Stmt(conn, &stmt);
error = SQLExec Direct(stmt, sqlquery, SQL_NTS);

if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR, branchname , 80, &lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,      0 , &lenOut2);

    while (SQLFetch(stmt) >= SQL_SUCCESS) {
        printf ("%s %g\n", branchname, balance);
    }
}
SQLFreeStmt(stmt, SQL_DROP);
```

More ODBC Features

I Prepared Statement

- || SQL statement prepared: compiled at the database
- || Can have placeholders: E.g. insert into account values(?, ?, ?)
- || Repeatedly executed with actual values for the placeholders

I Metadata features

- || finding all the relations in the database and
- || finding the names and types of columns of a query result or a relation in the database.

I By default, each SQL statement is treated as a separate transaction that is committed automatically.

- || Can turn off automatic commit on a connection
 - || SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
- || transactions must then be committed or rolled back explicitly by
 - || SQLTransact(conn, SQL_COMMIT) or
 - || SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
 - Core
 - Level 1 requires support for metadata querying
 - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

JDBC

- I JDBC is a Java API for communicating with database systems supporting SQL
- I JDBC supports a variety of features for querying and updating data, and for retrieving query results
- I JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- I Model for communicating with the database:
 - I Open a connection
 - I Create a “statement” object
 - I Execute queries using the Statement object to send queries and fetch results
 - I Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

JDBC Code (Cont.)

| Update to database

```
try {  
    stmt.executeUpdate( "insert into account values  
        ('A-9732', 'Perryridge', 1200)");  
}  
catch (SQLException sqle) {  
    System.out.println("Could not insert tuple. " + sqle);  
}
```

| Execute query and fetch and print results

```
ResultSet rset =stmt.executeQuery( "select branch_name, avg(balance)  
        from account  
        group by branch_name");  
  
while (rset.next()) {  
    System.out.println(  
        rset.getString("branch_name") +" " +rset.getFloat(2));  
}
```

JDBC Code Details

I Getting result fields:

 I **rs.getString("branchname") and rs.getString(1) equivalent if branchname is the first argument of select result.**

I Dealing with Null values

```
int a = rs.getInt("a");
if (rs.wasNull()) System.out.println("Got null value");
```



MODULE 4



Normalization: Database Design Theory

Introduction

- **Relation schema**
 - consists of a number of attributes
- **Relational database schema**
 - consists of a number of relation schemas
- **Assumption:**
 - attributes are grouped to form a relation schema by using the common sense of the database designer or
 - by mapping a database schema design from a conceptual data model such as the ER data model.
- These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations

- Database Design
 - coming up with a 'good' schema is very important
- How do we characterize the "goodness" of a schema ?
- If two or more alternative schemas are available how do we compare them ?
- What are the problems with "bad" schema designs ?
- This chapter discusses some of the theory that has been developed with the goal of evaluating relational schemas for design quality
 - to measure formally why one set of groupings of attributes into relation schemas is better than another.

- There are two levels at which we can discuss the goodness of relation schemas:

1. The **logical** (or **conceptual**) **level**—how users interpret the relation

schemas and the meaning of their attributes.

2. The **implementation** (or **physical storage**) **level**—how the tuples in a

base relation are stored and updated. This level applies only to

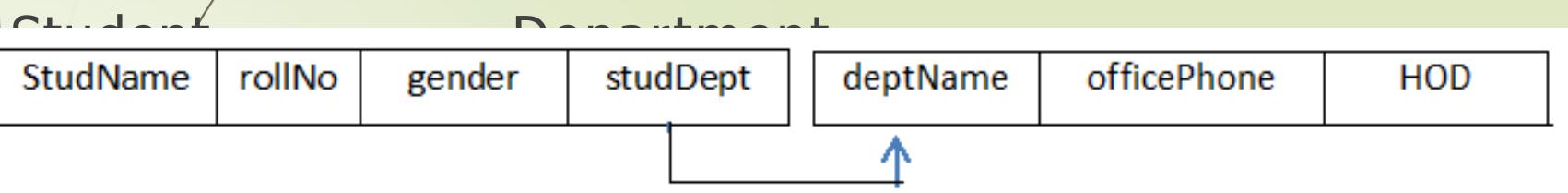
schemas of base relations

An Example

STUDENT relation with attributes: studName, rollNo, gender, studDept

- DEPARTMENT relation with attributes: deptName, officePhone, hod
- Several students belong to a department
- studDept gives the name of the student's department

Correct schema:



Incorrect schema:

~~student~~

StudName	rollNo	gender	deptName	officePhone	HOD
----------	--------	--------	----------	-------------	-----

- **Problems with bad schema**

- **Redundant storage of data:**

- Office Phone & HOD info -stored redundantly once with each

- student that belongs to the department

- wastage of disk space

- **A program that updates Office Phone of a department**

- must change it at several places
 - more running time
 - error -prone

Topics

Informal design guidelines for relation schemas

- criteria for good and bad relation schemas
- **Functional dependencies**
 - main tool for formally measuring the appropriateness of attribute groupings into relation schemas
- **Normal forms**
 - defined to meet a set of desirable constraints expressed using functional dependencies
- **Normalization**
 - applying a series of tests to relations to meet these increasingly stringent requirements and decompose the relations when necessary

Informal Design Guidelines for Relation Schemas

Four informal guidelines that may be used as measures to determine the quality of relation schema design:

1. Making sure that the semantics of the attributes is clear in the schema
 2. Reducing the redundant information in tuples
 3. Reducing the NULL values in tuples
 4. Disallowing the possibility of generating spurious tuples
- These measures are not always independent of one another

1. Imparting Clear Semantics to Attributes in Relations

- **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple
- 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 easier it is to explain the semantics of the relation, the better the relation schema design will be

EMPLOYEE

f.k.

ENAME	<u>SSN</u>	BDATE	ADDRESS	DNUMBER
p.k.				

DEPARTMENT

f.k.

DNAME	<u>DNUMBER</u>	DMGRSSN
p.k.		

DEPT_LOCATIONS

f.k.

DNUMBER	DLOCATION
p.k.	

PROJECT

f.k.

PNAME	<u>PNUMBER</u>	PLOCATION	DNUM
p.k.			

WORKS_ON

f.k.

f.k.

SSN	<u>PNUMBER</u>	HOURS
p.k.		

Fig 1: schema diagram for company

- schemas have a well-defined and unambiguous interpretation
- considered as easy to explain and therefore good from the standpoint of having clear semantics

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
 - if a relation schema corresponds to one entity type or one relationship type, it is straightforward to interpret and to explain its meaning
 - if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Violating Guideline 1

EMP_DEPT						
ENAME	<u>SSN</u>	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN

EMP_PROJ					
SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION

Fig 2: schema diagram for company

- Both the relation schemas have clear semantics
- A tuple in the EMP_DEPT relation schema represents a single employee but includes additional information—the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssN) of the department manager.

- A tuple in the EMP_PROJ relates an employee to a project but also includes the employee name (Ename), project name (Pname), and project location (Plocation)
- logically correct but they violate Guideline 1 by mixing attributes from distinct real-world entities:
 - EMP_DEPT mixes attributes of employees and departments
 - EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship
- They may be used as views, but they cause problems when used as

2. Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations

- Grouping attributes into relation schemas has a significant effect on storage space
- For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT with that for an EMP_DEPT base relation
- In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department
- In contrast, each department's information appears only once in the DEPARTMENT relation. Only the department number Dnumber is repeated in the EMPLOYEE relation for

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	gender	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	<u>Dnumber</u>	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

<u>Dnumber</u>	<u>Dlocation</u>
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

Figure :One possible database state for the COMPANY relational database schema (figure 1)

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	gender	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

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

Figure :One possible database state for the COMPANY relational database schema (figure 1)

EMP_DEPT

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

EMP_PROJ

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

Fig: Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 1

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

Insertion Anomalies

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

1. To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs

- For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT
- In the design of Employee in fig 1, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation

2. It is difficult to insert a new department that has no employees as yet in

the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee

- This violates the entity integrity for EMP_DEPT because Ssn is its primary key
- This problem does not occur in the design of Figure 1 because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies

The problem of deletion anomalies is related to the second insertion anomaly situation just discussed

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

Modification Anomalies

- In `EMP_DEPT`, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent
- If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong

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
 - The second guideline is consistent with and, in a way, a restatement of the first guideline
 - These guidelines may sometimes have to be violated in order to improve the performance of certain queries.

3. NULL Values in Tuples

If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples

- waste space at the storage level
- may also lead to problems with understanding the meaning of the attributes
- with specifying JOIN operations
- 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. For example, Visa_status may not apply to U.S. students.
 - The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
 - The value is *known but absent*; that is, it has not been recorded yet.

For example, the Home Phone Number for an employee

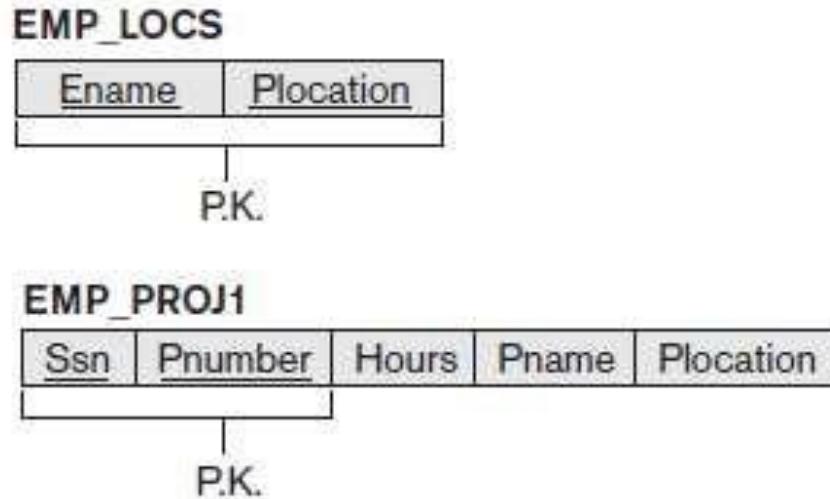
Guideline 3

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
- Using space efficiently and avoiding joins with NULL values are the two overriding criteria that determine whether to include the columns that may have NULLs in a relation or to have a separate relation for those columns with the appropriate key columns
- For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created to include tuples for only the employees with individual offices.

4 Generation of Spurious Tuples

- Consider the two relation schemas EMP_LOCS and EMP_PROJ1 which can be used instead of the single EMP



- A tuple in EMP_LOCS means that the employee whose name is Ename works on *some project* whose location is Plocation
- A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works Hours per week on the project whose name, number, and location are Pname,

EMP_LOCS

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

EMP_PROJ1

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

- Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS
- If we attempt a NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ
- Additional tuples that were not in EMP_PROJ are called **spurious tuples** because they represent spurious information that is not valid.
- The spurious tuples are marked by asterisks (*)

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

⋮

- Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because when we JOIN them back using NATURAL JOIN, we do not get the correct original information
- This is because in this case Plocation is the attribute that relates EMP_LOCS and EMP_PROJ1, and Plocation is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1.

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.

Summary and Discussion of Design Guidelines

we informally discussed situations that lead to problematic relation schemas and we proposed informal guidelines for a good relational design

- The problems we pointed out, which can be detected without additional tools of analysis, are as follows:
 - 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

Functional Dependencies

Formal tool for analysis of relational schemas that enables us to detect and describe some of the problems in precise terms

Definition of Functional Dependency

- A functional dependency is a constraint between two sets of attributes from the database.
- Given a relation R , a set of attributes X in R is said to **functionally determine** another attribute Y , also in R , (written $X \rightarrow Y$) if and only if each X value is associated with at most one Y value.
- X is the determinant set and Y is the dependent attribute. Thus, given a tuple and the values of the attributes in X , one can determine the corresponding value of the Y attribute.

- 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**.
- 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 to specify the functional dependencies that should hold on *all* relation states (extensions) r of R .

- Consider the relation schema EMP_PROJ;

EMP_PROJ					
SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION

- From the semantics of the attributes and the relation, we know that the following functional dependencies should hold:
 - $\text{Ssn} \rightarrow \text{Ename}$
 - $\text{Pnumber} \rightarrow \{\text{Pname}, \text{Plocation}\}$
 - $\{\text{Ssn}, \text{Pnumber}\} \rightarrow \text{Hours}$

- These functional dependencies specify that

(a) the value of an employee's Social Security number
(Ssn) uniquely

determines the employee name (Ename)

(b) the value of a project's number (Pnumber) uniquely
determines

the project name (Pname) and location (Plocation),
and
(c) a combination of Ssn and Pnumber values uniquely
determines

the number of hours the employee currently works on

the

- Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R
- 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

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

TEACHER → COURSE

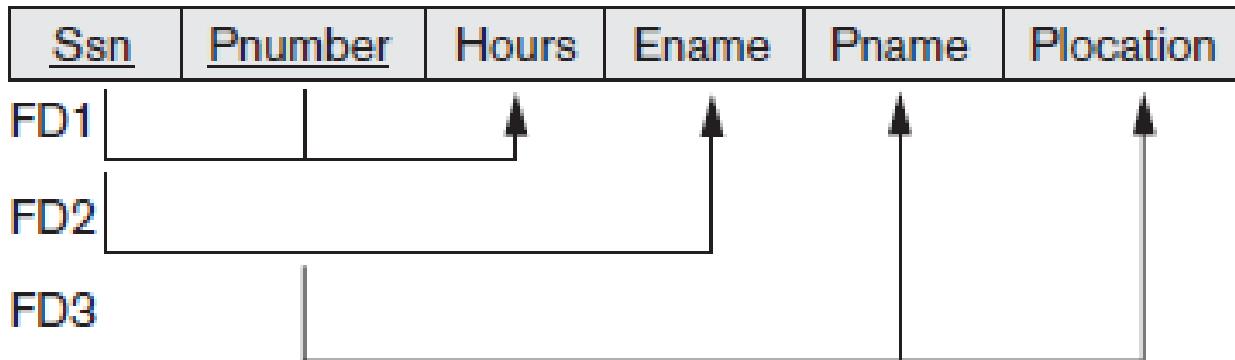
Example:



A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

1. $B \rightarrow C$
2. $C \rightarrow B$
3. $\{A, B\} \rightarrow C$
4. $\{A, B\} \rightarrow D$
5. $\{C, D\} \rightarrow B.$
6. $A \rightarrow B$
7. $B \rightarrow A$
8. $D \rightarrow C$

EMP_PROJ



- 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
- The right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

Normal Forms Based on Primary Keys

- Set of functional dependencies is given for each relation
- Each relation has a designated primary key
- This information combined with the tests (conditions) for normal forms drives the normalization process for relational schema design
- First three normal forms for relation
 - Takes into account all candidate keys of a relation rather than just the primary key

Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**.

- Initially, Codd proposed three normal forms, which he called first, second, and third normal form
- All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation
- A fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively

- **Normalization of data** can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of
 - (1) minimizing redundancy and
 - (2) minimizing the insertion, deletion, and update anomalies
- It can be considered as a “filtering” or “purification” process to make the design have successively better quality
- Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties.

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

Practical Use of Normal Forms

- Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- Database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.
- The database designers *need not* normalize to the highest possible normal form
- Relations may be left in a lower normalization status, such as 2NF, for performance reasons

Definitions of Keys and Attributes Participating in Keys

Superkey: specifies a uniqueness constraint that no two distinct tuples in any state r of R can have the same value

- **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more
- Example:
 - The attribute set {Ssn} is a key because no two employees tuples can have the same value for Ssn
 - Any set of attributes that includes Ssn—for example, {Ssn, Name, Address)—is a superkey
- If a relation schema has more than one key, each is called a **candidate key**
- One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called **secondary keys**

- In a practical relational database, each relation schema must have a primary key
 - If no candidate key is known for a relation, the entire relation can be treated as a default superkey
- For example {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key
- **Definition.** An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R . An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key

WORKS_ON		
F.K.	F.K.	
Ssn	Pnumber	Hours
P.K.		

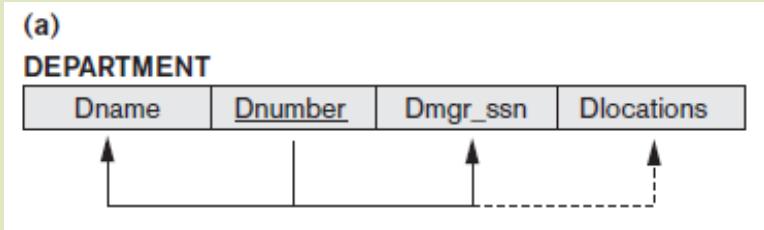
- In WORKS_ON relation Both Ssn and Pnumber are prime attributes
 - whereas other attributes are nonprime.

First Normal Form

Defined to disallow multivalued attributes, composite attributes, and their combinations

- It states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute
- 1NF disallows relations within relations or relations as attribute values within tuples
- The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

- Consider the DEPARTMENT relation schema shown in Figure below



- Primary key is Dnumber
- We assume that each department can have a number of locations
- The DEPARTMENT schema and a sample relation state are shown in Figure below

DEPARTMENT

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

- As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure

- There are two ways we can look at the Dlocations attribute:
 - The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber
 - The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber→Dlocations because each set is considered a single member of the attribute domain
- In either case, the DEPARTMENT relation is not in 1NF

There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and 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}. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

2. Expand the key so that there will be a separate tuple in the original

DEPARTMENT relation for each location of a DEPARTMENT. In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation



DEPARTMENT

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

3. If a maximum number of values is known for the attribute—for example,

if it is known that at most three locations can exist for a department— replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing NULL values if most departments have fewer than three locations. Querying on this attribute becomes more difficult; for example,

- Of the three solutions, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values
- First normal form also disallows multivalued attributes that are themselves composite.
- These are called **nested relations** because each tuple can have a relation within it.

(a)

EMP_PROJ		Projs	
Ssn	Ename	Pnumber	Hours

- Figure above shows how the EMP_PROJ relation could appear if nesting is allowed
- Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee's projects and the hours per week that employee works on each project.
- The schema of this EMP_PROJ relation can be represented as follows:

- Ssn is the primary key of the EMP_PROJ relation and Pnumber is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber
- To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation
- Decomposition and primary key propagation yield the schema

EMP_PROJ1	
Ssn	Ename

EMP_PROJ2		
Ssn	Pnumber	Hours

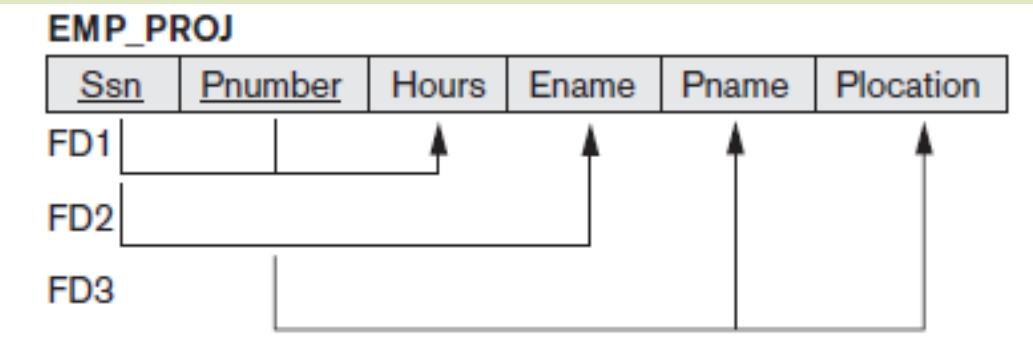
EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	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

Second Normal Form

Second normal form (2NF) is based on the concept of full functional dependency

- A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y
- A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$

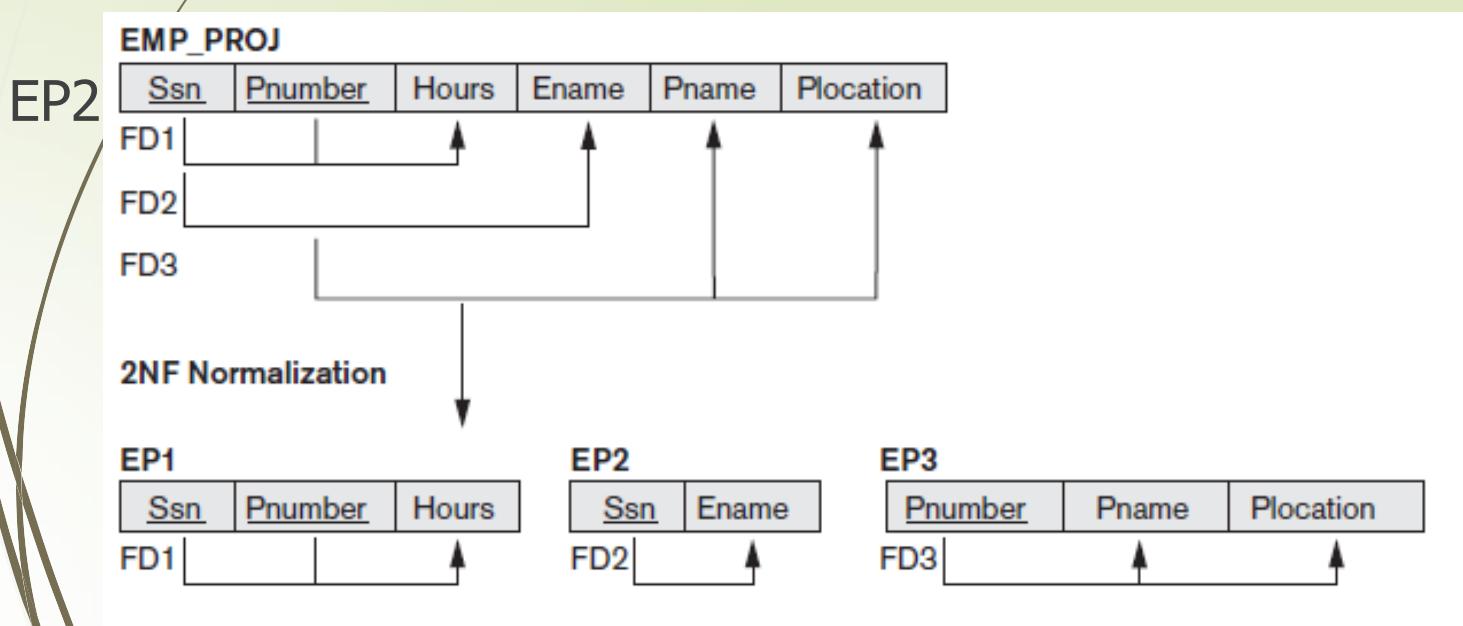


- In the above figure , $\{Ssn, Pnumber\} \rightarrow Hours$ is a full dependency (neither $Ssn \rightarrow Hours$ nor $Pnumber \rightarrow Hours$ holds)
- $\{Ssn, Pnumber\} \rightarrow Ename$ is partial because $Ssn \rightarrow Ename$ holds

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

- The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key
- If the primary key contains a single attribute, the test need not be applied at all
- The EMP_PROJ relation is in 1NF but is not in 2NF.
- The nonprime attribute Ename violates 2NF because of FD2,
- as do the nonprime attributes Pname and Plocation because of FD3
- The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.

- If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which **nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent.**
- Therefore, the functional dependencies FD1, FD2, and FD3 lead to the decomposition of EMP_PROJ into the three relation schemas EP1,

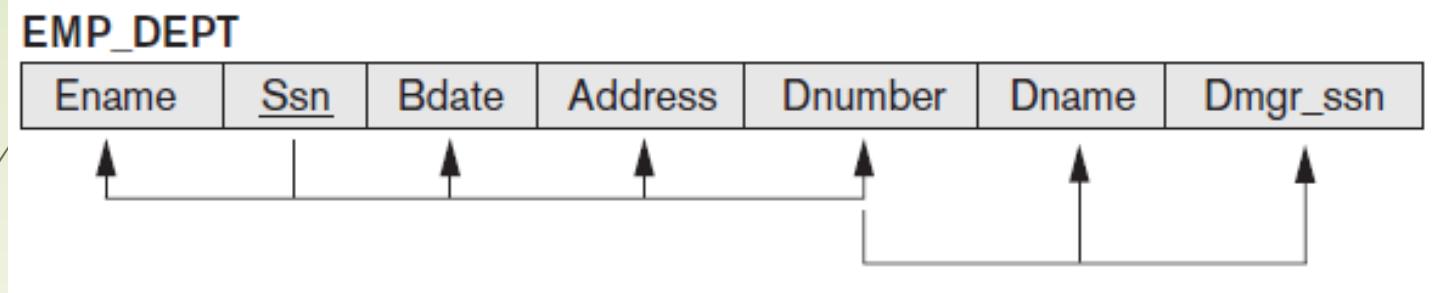


Third Normal Form

Transitive functional dependency

A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attribute Z that are neither a primary nor a subset of any key of R (candidate key) and both $X \rightarrow Z$ and $Y \rightarrow Z$ holds

Example:



■ **SSN → DMGRSSN is a transitive FD** since

SSN → DNUMBER and DNUMBER → DMGRSSN hold

Dnumber is neither a key itself nor a subset of the key of EMP_DEPT

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

Definition:

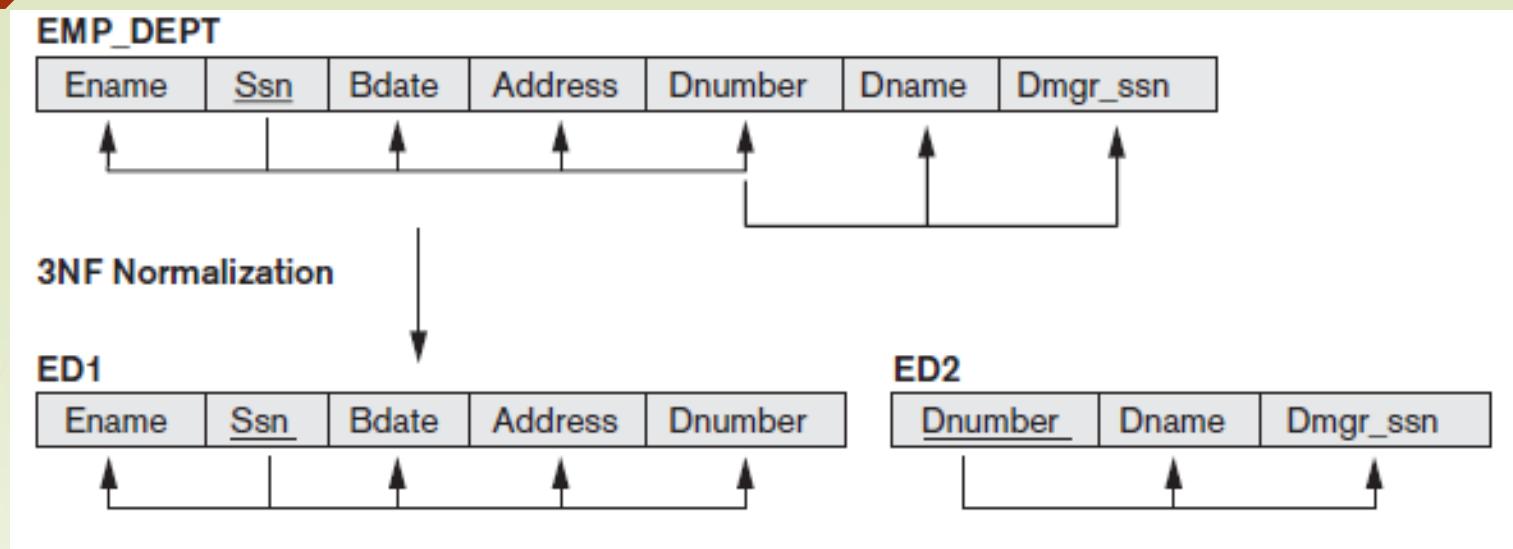
A relation schema R is in third normal form (3NF) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key

- The relation schema EMP_DEPT is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn
(and Ssn)

EMP_DEPT						
Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn

```
graph TD; Ename --> Dnumber; Ssn --> Dnumber; Bdate --> Dnumber; Address --> Dnumber; Dnumber --> Dname; Dnumber --> Dmgr_ssn;
```

- We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2



- ED1 and ED2 represent independent entity facts about employees and departments
- A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples

- Problematic FD
 - Left-hand side is part of primary key
 - Left-hand side is a non-key attribute
- 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations
- In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and 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.
Third (3NF)	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.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

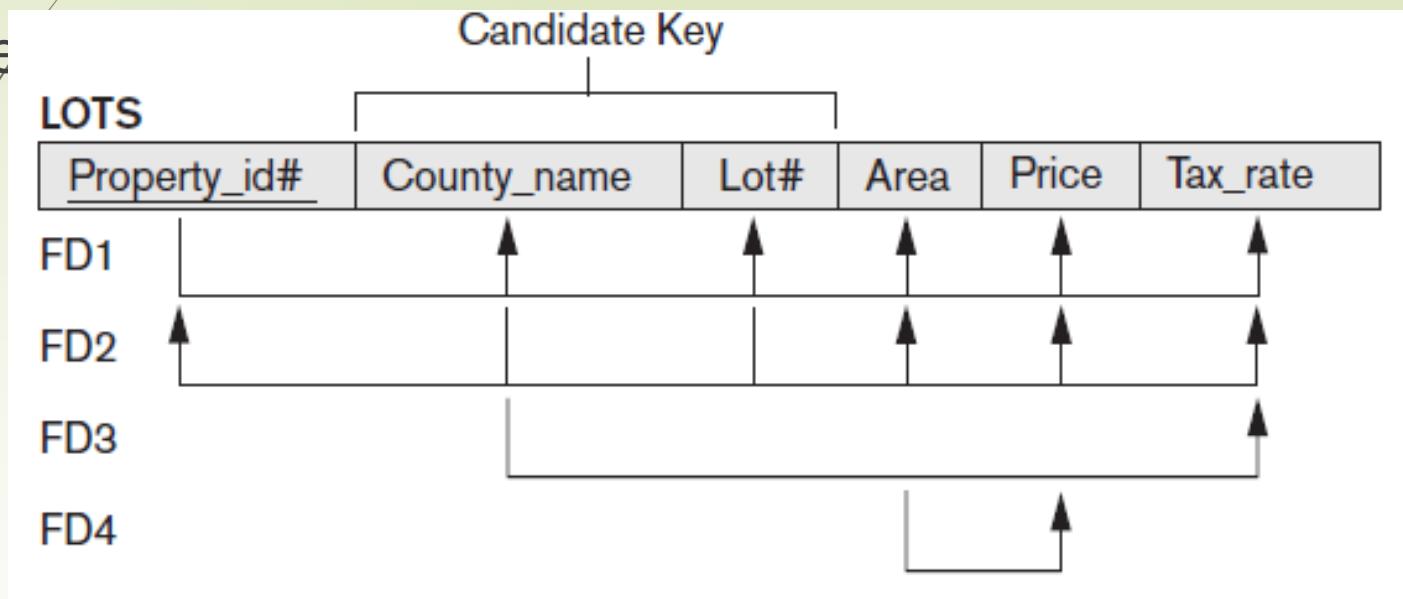
General Definition of Second and Third Normal Form

Takes into account all candidate keys of a relation into account

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

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

- Consider the relation schema LOTS which describes parcels of land for sale in various counties of a state
- Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.



- Based on the two candidate keys `Property_id#` and `{County_name, Lot#}`, the functional dependencies FD1 and FD2 hold
 - FD1: $\text{Property_id} \rightarrow \{\text{County_name}, \text{Lot\#}, \text{Area}, \text{Price}, \text{Tax_rate}\}$
 - FD2: $\{\text{County_name}, \text{Lot\#}\} \rightarrow \{\text{Property_id}, \text{Area}, \text{Price}, \text{Tax_rate}\}$
 - FD3: $\text{County_name} \rightarrow \text{Tax_rate}$
 - FD4: $\text{Area} \rightarrow \text{Price}$
- We choose `Property_id#` as the primary key, but no special consideration will be given to this key over the other candidate key
- FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county)
- FD4 says that the price of a lot is determined by its area regardless of which county it is in.

- The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3
- To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2

LOTS1

Property_id#	County_name	Lot#	Area	Price
FD1				
FD2				
FD4				

LOTS2

County_name	Tax_rate
FD3	

- We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2.
- Both LOTS1 and LOTS2 are in 2NF.

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

- According to this definition, LOTS2 is in 3NF
- FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1
- To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B

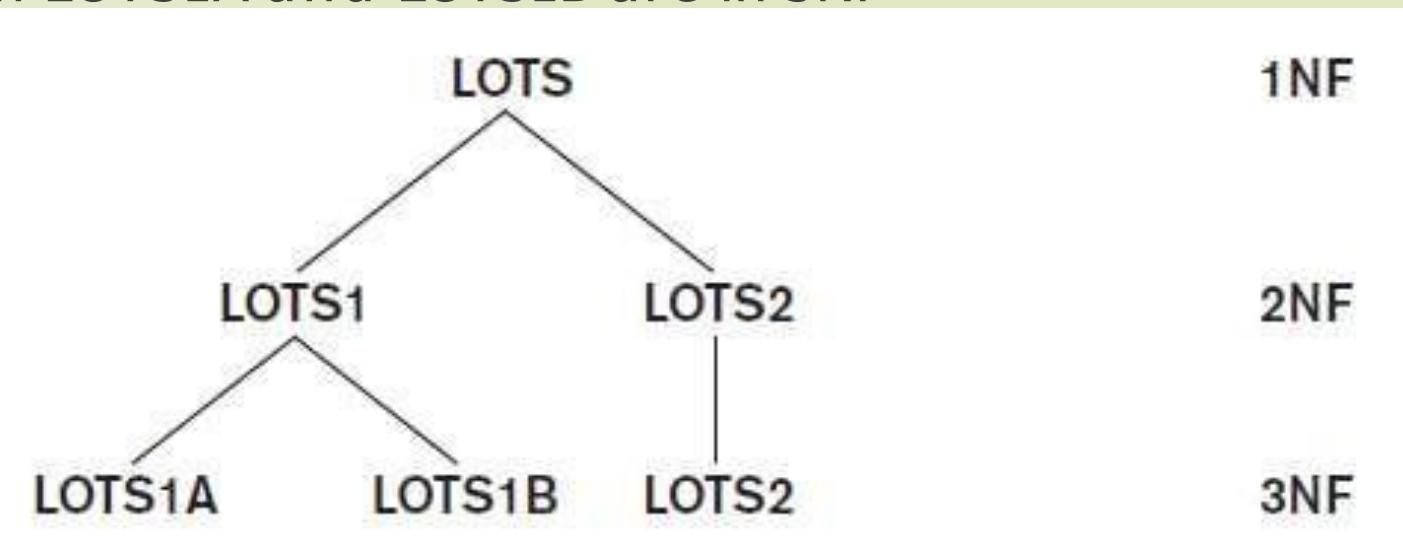
LOTS1A

<u>Property_id#</u>	County_name	Lot#	Area
FD1			
FD2			

LOTS1B

Area	Price
FD4	

- We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the lefthand side of FD4 that causes the transitive dependency) into another relation LOTS1B.
- Both LOTS1A and LOTS1B are in 3NF



Boyce-Codd Normal Form

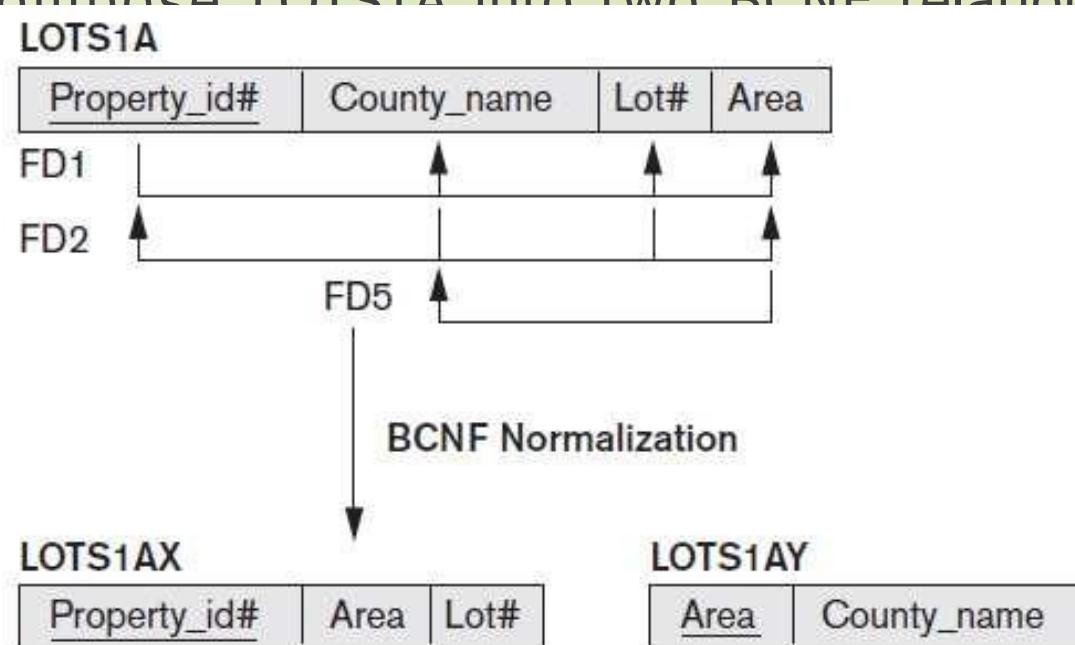
Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF

- Every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF
- Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton
- Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to
 1.1, 1.2, ..., 1.9, and 2.0 acres
- In such a situation we would have the additional functional dependency FD5: $\text{Area} \rightarrow \text{County_name}$
- If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a

- The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation $R(\text{Area}, \text{County_name})$, since there are only 16 possible Area values
- This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples
- BCNF is a stronger normal form that would disallow LOTS1A and suggest the need for decomposing it

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

- The formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF
 - In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A
 - FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF
 - We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY
- ~~dependency exists in the same relation~~



- In practice, most relation schemas that are in 3NF are also in BCNF
- Only if $X \rightarrow A$ 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
- Example: consider the relation TEACH with the following dependencies

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
Narayan	Operating Systems	Ammar

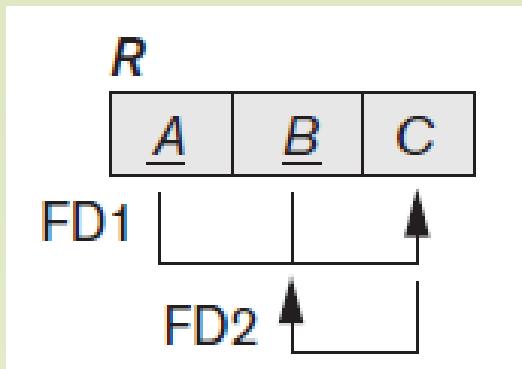
FD1: {Student, Course} → Instructor

FD2: Instructor → Course -- means that each instructor teaches one

course

- {Student, Course} is a candidate key for this relation

The dependencies shown follow the pattern in Figure below with Student as *A*, Course as *B*, and Instructor as *C*



- Hence this relation is in 3NF but not BCNF
- Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:
 1. R1(Student, Instructor) and R2(Student, Course)
 2. R1(Course, Instructor) and R2(Course, Student)
 3. R1(Instructor, Course) and R2(Instructor, Student)

- It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF
- Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:
 - The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
 - The **dependency preservation property**, which ensures that

- We are not able to meet the functional dependency preservation ,but we must meet the non additive join property

- **Nonadditive Join Test for Binary Decomposition:**

A decomposition $D=\{R_1, R_2\}$ of R has the lossless join property with respect to a set of functional dependencies F on R if and only if either

- The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+ or
 - The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+
 - The third decomposition meets the test
- $R_1 \cap R_2$ is Instructor
- $R_1 - R_2$ is Course
- Hence, the proper decomposition of TEACH into BCNF relations is:
TEACH1(Instructor,Course) and TEACH2(Instructor,Student)

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 set of relations that are in BCNF:



Let R be the relation not in BCNF, let X ⊆ R, and let X → A be the FD that causes 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

Multivalued Dependency and Fourth Normal Form

For example, consider the relation EMP shown in Figure below:

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

- A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname
- An employee may work on several projects and may have several dependents
- The employee's projects and dependents are independent of one another

- To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project
- In the relation state shown in the EMP, the employee Smith works on two projects 'X' and 'Y' and has two dependents 'John' and 'Anna' and therefore there are 4 tuples to represent these facts together
- The relation EMP is an **all-key relation** (with key made up of all attributes) and therefore no f.d's and as such qualifies to be a BCNF relation
- There is a redundancy in the relation EMP-the dependent information is repeated for every project and project information is repeated for every dependent

- To address this situation, the concept of multivalued dependency(MVD) was proposed and based on this dependency, the fourth normal form was defined
- Multivalued dependencies are a consequence of 1NF which disallows an attribute in a tuple to have a set of values, and the accompanying process of converting an unnormalized relation into 1NF
- Informally, whenever two independent 1:N relationships are mixed in the same relation, $R(A, B, C)$, an MVD may arise

Formal Definition of Multivalued Dependency

Definition. A multivalued dependency $X \rightarrow\!\!\! \rightarrow Y$ specified on relation schema R , where X and Y are both subsets of R , specifies the following constraint on any relation state r of R : If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties where we use Z to denote $(R - (X \cup Y))$

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

EMP

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

t1[Ename]=t3[Ename], Y=Pname
t2[Ename]=Smith
 $\neg(t_1[\text{Ename}] \cup t_2[\text{Ename}])$

- | t3(Ename)=t4(Ename)=t1(Ename)=t2(Ename)=Smith
- | t3(Pname)=t1(Pname)=X and t4(Pname)=t2(Pname)=Y
- | t3(Dname)=t2(Dname)=Anna and
t4(Dname)=t1(Dname)=John
- Whenever $X \rightarrow\rightarrow Y$ holds, we say that **X multidetermines Y**.
Because of the symmetry in the definition, whenever $X \rightarrow\rightarrow Y$ holds in R , so does $X \rightarrow\rightarrow Z$. Hence, $X \rightarrow\rightarrow Y$ implies $X \rightarrow\rightarrow Z$, and therefore it is sometimes written as $X \rightarrow\rightarrow Y \mid Z$

- An MVD $X \rightarrow\!\!\!\rightarrow Y$ in R is called a **trivial MVD** if

(a) Y is a subset of X , or

(b) $X \cup Y = R$

EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

- For example, the relation EMP_PROJECTS has the trivial MVD
 $\text{Ename} \rightarrow\!\!\!\rightarrow \text{Pname}$
- An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**
- If we have a *nontrivial MVD* in a relation, we may have to repeat values redundantly in the tuples
- In the EMP relation the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname)

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

Definition: A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every nontrivial multivalued dependency $X \twoheadrightarrow Y$ in F^+X is a superkey for R

- The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD

EMP_PROJECTS

<u>Ename</u>	<u>Pname</u>
Smith	X
Smith	Y

EMP_DEPENDENTS

<u>Ename</u>	<u>Dname</u>
Smith	John
Smith	Anna

- We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS
- Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs $Ename \rightarrow\!\!\rightarrow Pname$ in EMP_PROJECTS and $Ename \rightarrow\!\!\rightarrow Dname$ in EMP_DEPENDENTS are trivial MVDs
- No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either

- We can state the following points:
 - An all-key relation is always in BCNF since it has no FDs
 - An all-key relation such as the EMP, which has no FDs but has the MVD $\text{Ename} \rightarrow\rightarrow \text{Pname} \mid \text{Dname}$, is not in 4NF
 - A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF
 - The decomposition removes the redundancy caused by the MVD

Join Dependencies and Fifth Normal Form

A **join dependency (JD)**, denoted by $\text{JD}(R_1, R_2, \dots, R_n)$, specified on relation schema R , specifies a constraint on the states r of R . The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, \dots, R_n . Hence, for every such r we have

$$*(\pi_{R_1}(r), \pi_{R_2}(r), \dots, \pi_{R_n}(r)) = r$$

- A join dependency $\text{JD}(R_1, R_2, \dots, R_n)$, specified on relation schema R , is a **trivial JD** if one of the relation schemas R_i in $\text{JD}(R_1, R_2, \dots, R_n)$ is equal to R .

Fifth normal form(project-join normal form)

A relation schema R is in **fifth normal form (5NF)** (or **project-join normal form (PJNF)**) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency $\text{JD}(R_1, R_2, \dots, R_n)$ in F^+ every R_i is a superkey of R .

- A database is said to be in 5NF, if and only if,
 - It's in 4NF
 - If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records.

SUPPLY

<u>Sname</u>	<u>Part_name</u>	<u>Proj_name</u>
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

Fig: The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R_1, R_2, R_3)

R_1

<u>Sname</u>	<u>Part_name</u>
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

<u>Sname</u>	<u>Proj_name</u>
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

<u>Part_name</u>	<u>Proj_name</u>
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Fig: Decomposing the relation SUPPLY into the 5NF relations R_1, R_2, R_3 .

Further Topics in Functional Dependencies

- Functional dependencies and show how new dependencies can be inferred from a given set
- The concepts of closure, equivalence, and minimal cover

Inference Rules for Functional Dependencies

- F the set of functional dependencies that are specified on relation schema R
- The schema designer specifies the functional dependencies that are semantically obvious
- Numerous other functional dependencies hold in all legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F
- Those other dependencies can be inferred or deduced from the FDs in F.

For example:

If each department has one manager, so that Dept_no uniquely determines Mgr_ssn ($\text{Dept_no} \rightarrow \text{Mgr_ssn}$), and a manager has a unique phone number called Mgr_phone ($\text{Mgr_ssn} \rightarrow \text{Mgr_phone}$),

- Then these two dependencies together imply that
 $\text{Dept_no} \rightarrow \text{Mgr_phone}$
- This is an **inferred FD** and need *not* be explicitly stated in addition to the two given FDs.

Definition. Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the **closure** of F ; it is denoted by F^+ .

- For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema EMP_DEPT

EMP_DEPT

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

$$F = \{$$

$\text{Ssn} \rightarrow \{\text{Ename}, \text{Bdate}, \text{Address}, \text{Dnumber}\},$

$\text{Dnumber} \rightarrow \{\text{Dname}, \text{Dmgr_ssn}\}$

}

- Some of the additional functional dependencies that we can *infer* from F are the following:
 - $\text{Ssn} \rightarrow \{\text{Dname}, \text{Dmgr_ssn}\}$
 - $\text{Ssn} \rightarrow \text{Ssn}$
 - $\text{Dnumber} \rightarrow \text{Dname}$
- An FD $X \rightarrow Y$ is **inferred from** a set of dependencies F specified on R if $X \rightarrow Y$ holds in every legal relation state r of R
- The closure F^+ of F is the set of all functional dependencies that can be inferred from F

- Set of **inference rules** can be used to infer new dependencies from a given set of dependencies
- We use the notation $F \Rightarrow X \rightarrow Y$ to denote that the functional dependency $X \rightarrow Y$ is inferred from the set of functional dependencies F
- we use an abbreviated notation when discussing functional dependencies. We concatenate attribute variables and drop the commas for convenience
- The FD $\{X, Y\} \rightarrow Z$ is abbreviated to $XY \rightarrow Z$, and the FD $\{X, Y, Z\} \rightarrow \{U, V\}$ is abbreviated to $XYZ \rightarrow UV$.

- Three rules IR1 through IR3 are well-known inference rules for functional dependencies.
- They are proposed by Armstrong and hence known as **Armstrong's axioms**

- IR1 (reflexive rule): If $X \supseteq Y$, then $X \rightarrow Y$.
- IR2 (augmentation rule): $\{X \rightarrow Y\} \cup XZ \rightarrow YZ$.
- IR3 (transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} \cup X \rightarrow Z$
- The reflexive rule (IR1) states that a set of attributes always determines itself or any of its subsets, which is obvious.
- Because IR1 generates dependencies that are always true, such dependencies are called *trivial*.
- Formally, a functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**.

- The augmentation rule (IR2) says that adding the same set of attributes to both the left- and right-hand sides of a dependency

results in another valid dependency

- According to IR3, functional dependencies are transitive
- There are three other inference rules that follow from IR1, IR2 and IR3. They are:
 - IR4 (decomposition, or projective, rule): $\{X \rightarrow YZ\} \Rightarrow X \rightarrow Y$
 - IR5 (union, or additive, rule): $\{X \rightarrow Y, X \rightarrow Z\} \Rightarrow X \rightarrow YZ$
 - IR6 (pseudotransitive rule): $\{X \rightarrow Y, WY \rightarrow Z\} \Rightarrow WX \rightarrow Z$

- The decomposition rule (IR4) says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD $X \rightarrow \{A_1, A_2, \dots, A_n\}$ into the set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$.
- The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$ into the single FD $X \rightarrow \{A_1, A_2, \dots, A_n\}$.
- The pseudotransitive rule (IR6) allows us to replace a set of attributes Y on the left hand side of a dependency with another set X that functionally determines Y , and can be derived from IR2 and IR3 if we augment the first functional dependency $X \rightarrow Y$ with W (the augmentation rule) and then

- It has been shown by Armstrong (1974) that inference rules IR1 through IR3 are sound and complete.
- By **sound**, we mean that given a set of functional dependencies specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds in every relation state r of R that satisfies the dependencies in F.
- By **complete**, we mean that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of all possible dependencies that can be inferred from F.
- In other words, the set of dependencies F^+ , which we called the **closure** of F, can be determined from F by using only

- A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of *all attributes* that are dependent on X .
- **Definition.** For each such set of attributes X , we determine the set X^+ of attributes that are functionally determined by X based on F ; X^+ is called the **closure of X under F** .
- Algorithm 16.1 can be used to calculate X^+ .

Algorithm 16.1. Determining X^+ , the Closure of X under F

Input: A set F of FDs on a relation schema R , and a set of attributes X , which is a subset of R .

$X^+ := X;$

repeat

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

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

 if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z;$

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

- Algorithm 16.1 starts by setting X^+ to all the attributes in X .
- By IR1, we know that all these attributes are functionally dependent on X .
- Using inference rules IR3 and IR4, we add attributes to X^+ , using each functional dependency in F .
- We keep going through all the dependencies in F (the repeat loop) until no more attributes are added to X^+ during a complete cycle (of the for loop) through the dependencies in F .

- For example, consider the relation schema EMP_PRO. From the semantics of the attributes, we specify the following set F of functional dependencies that should hold on EMP_PRO:

$F = \{ Ssn \rightarrow Ename,$

$Pnumber \rightarrow \{ Pname, Plocation \},$

$\{ Ssn, Pnumber \} \rightarrow Hours \}$

- Using Algorithm 16.1, we calculate the following closure sets with respect to F :

□ $\{ Ssn \}^+ = \{ Ssn, Ename \}$

□ $\{ Pnumber \}^+ = \{ Pnumber, Pname, Plocation \}$

Equivalence of Sets of Functional Dependencies

Definition: A set of functional dependencies F is said to **cover** another set of functional dependencies E if every FD in E is also in F^+ ; that is, if every dependency in E can be inferred from F; alternatively, we can say that E is **covered by** F.

Definition: Two sets of functional dependencies E and F are **equivalent** if

$E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions—E covers F and F covers E—hold.

Minimal Sets of Functional Dependencies

A set of functional dependencies F to be **minimal** if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.

2. We cannot replace any dependency $X \rightarrow A$ in F with a dependency

$Y \rightarrow A$, where Y is a proper subset of X , and still have a set of

dependencies that is equivalent to F .

Extraneous attribute: An attribute in a functional dependency is considered an extraneous attribute if we can remove it without changing the closure of the set of dependencies.

Algorithm 16.2. Finding a Minimal Cover F for a Set of Functional Dependencies E

Input: A set of functional dependencies E .

1. Set $F := E$.
2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \dots, A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$.
3. For each functional dependency $X \rightarrow A$ in F
 for each attribute B that is an element of X
 if $\{F - \{X \rightarrow A\}\} \cup \{(X - \{B\}) \rightarrow A\}$ is equivalent to F
 then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F .
4. For each remaining functional dependency $X \rightarrow A$ in F
 if $\{F - \{X \rightarrow A\}\}$ is equivalent to F ,
 then remove $X \rightarrow A$ from F .

- Step 2 places FDs in a canonical form for subsequent testing
- Step 3 constitutes removal of an extraneous attribute B contained in the left-hand side X of a functional dependency $X \rightarrow A$ from F when possible
- Step 4 constitutes removal of a redundant functional

Example 1: Let the given set of FDs be $E: \{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$. We have to find the minimal cover of E .

- All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm and can proceed to step 2
- In step 2 we need to determine if $AB \rightarrow D$ has any redundant attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ or $A \rightarrow D$?
- Since $B \rightarrow A$, by augmenting with B on both sides (IR2), we have $BB \rightarrow AB$, or $B \rightarrow AB$ (i). However, $AB \rightarrow D$ as given (ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), $B \rightarrow D$. Thus $AB \rightarrow D$ may be replaced by $B \rightarrow D$.

- We now have a set equivalent to original E , say $E: \{B \rightarrow A, D \rightarrow A, B \rightarrow D\}$. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.
- In step 3 we look for a redundant FD in E . By using the transitive rule on $B \rightarrow D$ and $D \rightarrow A$, we derive $B \rightarrow A$. Hence $B \rightarrow A$ is redundant in E and can be eliminated.
- Therefore, the minimal cover of E is $\{B \rightarrow D, D \rightarrow A\}$.

Example 2: Let the given set of FDs be $G: \{A \rightarrow BCDE, CD \rightarrow E\}$. Find the minimal cover of G

Algorithm 16.2(a). Finding a Key K for R Given a set F of Functional Dependencies

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

1. Set $K := R$.
2. For each attribute A in K

{compute $(K - A)^+$ with respect to F ;

if $(K - A)^+$ contains all the attributes in R , then set $K := K - \{A\}$ };

- We start by setting K to all the attributes of R ; we then remove one attribute at a time and check whether the remaining attributes still form a superkey.
- Algorithm 16.2(a) determines only *one* key out of the possible candidate keys for R ; the key returned depends on the order in which attributes are removed from R in step 2.

Properties of Relational Decompositions

Universal relation schema

- **Universal relation schema** $R = \{A_1, A_2, \dots, A_n\}$ includes all the attributes of the database
- **universal relation assumption:** 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
- 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 .

Attribute Preservation condition of a Decomposition

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

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

- 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

Desirable Properties of Decompositions

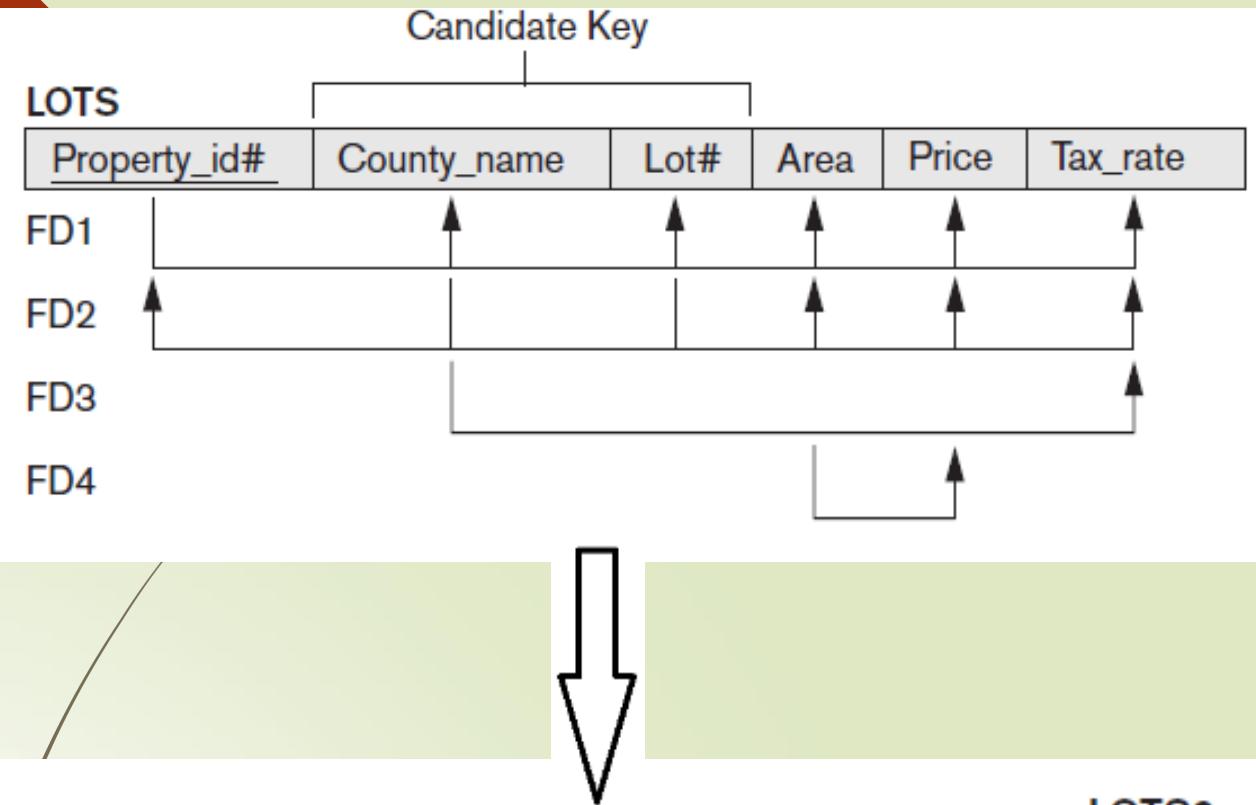
- Not all decomposition of a schema are useful
- We require two properties to be satisfied:
 - i) Dependency Preservation Property
 - ii) Nonadditive (Lossless) Join Property

Dependency Preservation Property

Each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i .

- 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 R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation
- 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

Example: Dependency Preserving Decomposition



LOTS1

Property_id#	County_name	Lot#	Area	Price
--------------	-------------	------	------	-------

FD1

FD2

FD4

LOTS2

County_name	Tax_rate
-------------	----------

FD3

Example: Decomposition that does not Preserve Dependency

LOTS1A

Property_id#	County_name	Lot#	Area
--------------	-------------	------	------

FD1

FD2

FD5

BCNF Normalization

LOTS1AX

Property_id#	Area	Lot#
--------------	------	------

LOTS1AY

Area	County_name
------	-------------

Nonadditive (Lossless) Join Property

Ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the 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 (Π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous or invalid information
- The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations
- The term **lossy design** refer to a design that represents a loss of information

Algorithm 16.3. Testing for Nonadditive Join Property

Input: A universal relation R , a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R , and a set F of functional dependencies.

Note: Explanatory comments are given at the end of some of the steps. They follow the format: (* comment *).

1. Create an initial matrix S with one row i for each relation R_i in D , and one column j for each attribute A_j in R .
2. Set $S(i, j) := b_{ij}$ for all matrix entries. (* each b_{ij} is a distinct symbol associated with indices (i, j) *).
3. For each row i representing relation schema R_i
{for each column j representing attribute A_j
{if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$;};}; (* each a_j is a distinct symbol associated with index (j) *).
4. Repeat the following loop until a *complete loop execution* results in no changes to S
{for each functional dependency $X \rightarrow Y$ in F
{for all rows in S that have the same symbols in the columns corresponding to attributes in X
{make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an a symbol for the column, set the other rows to that same a symbol in the column. If no a symbol exists for the attribute in any of the rows, choose one of the b symbols that appears in one of the rows for the attribute and set the other rows to that same b symbol in the column ;} ;} ;};
5. If a row is made up entirely of a symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

- (a) $R = \{\text{Ssn, Ename, Pnumber, Pname, Plocation, Hours}\}$ $D = \{R_1, R_2\}$
- $R_1 = \text{EMP_LOCS} = \{\text{Ename, Plocation}\}$
- $R_2 = \text{EMP_PROJ1} = \{\text{Ssn, Pnumber, Hours, Pname, Plocation}\}$
- $F = \{\text{Ssn} \rightarrow\!\!> \text{Ename}; \text{Pnumber} \rightarrow\!\!> \{\text{Pname, Plocation}\}; \{\text{Ssn, Pnumber}\} \rightarrow\!\!> \text{Hours}\}$

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(No changes to matrix after applying functional dependencies)

(b)	EMP	PROJECT	WORKS_ON
	Ssn Ename	Pnumber Pname Plocation	Ssn Pnumber Hours

- (c) $R = \{Ssn, Ename, Pnumber, Pname, Plocation, Hours\}$ $D = \{R_1, R_2, R_3\}$
- $R_1 = EMP = \{Ssn, Ename\}$
- $R_2 = PROJ = \{Pnumber, Pname, Plocation\}$
- $R_3 = WORKS_ON = \{Ssn, Pnumber, Hours\}$

$$F = \{Ssn \rightarrow Ename; Pnumber \rightarrow (Pname, Plocation); (Ssn, Pnumber) \rightarrow Hours\}$$

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(Original matrix S at start of algorithm)

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32} a_2	a_3	b_{34} a_4	b_{35} a_5	a_6

(Matrix S after applying the first two functional dependencies;
last row is all "a" symbols so we stop)

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and 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.
Third (3NF)	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.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

Problem 1

Consider the following relation for published books:

BOOK(BookTitle, AuthorName, BookType, ListPrice, AuthorAffiliation, Publisher)

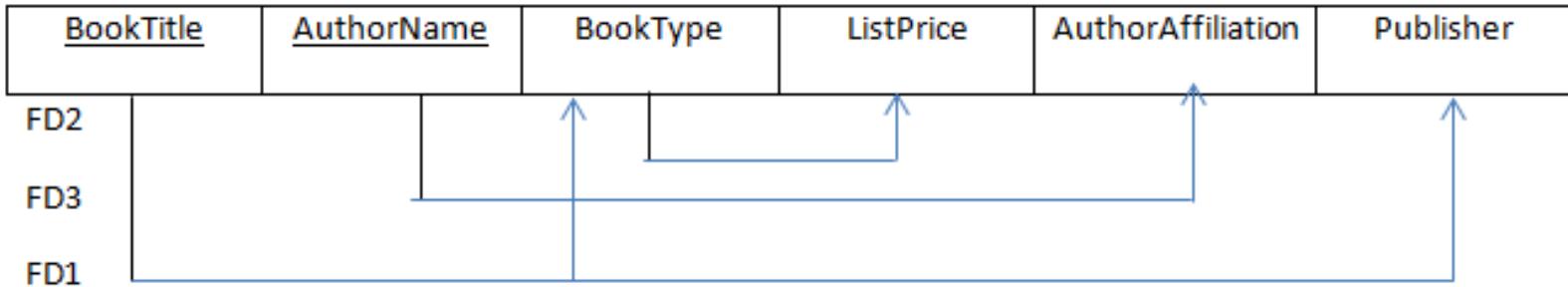
Suppose the following dependencies exist:

- | **BookTitle → BookType, Publisher**
- | **BookType → ListPrice**
- | **AuthorName → AuthorAffiliation**

What normal form is the relation in? explain your answer. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

Solution:

The relation is in 1NF and not in 2NF as no attributes are fully functionally dependent on the key (BookTitle and AuthorName). It is also not in 3NF.



| The relation is not in 2NF because the partial Dependencies exist

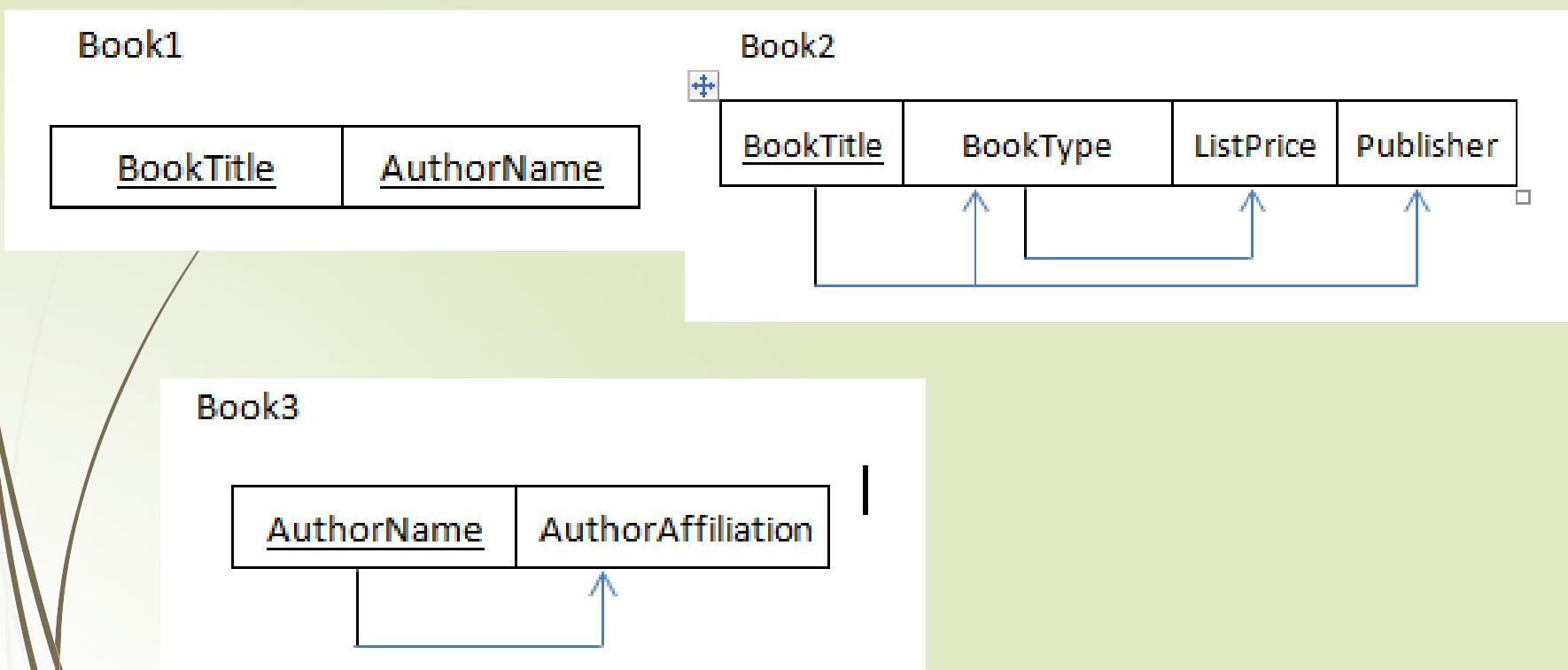
$\{BookTitle, AuthorName\} \rightarrow \{Publisher, BookType\}$

$\{BookTitle, AuthorName\} \rightarrow AuthorAffiliation$

| Thus, these attributes are not fully functionally dependent on the primary key. The 2NF decomposition will eliminate the partial dependencies.

- 2NF decomposition:

- Book1(BookTitle, AuthorName)
- Book2(BookTitle, BookType, ListPrice, Publisher)
- Book3(AuthorName, AuthorAffiliation)



- The relations are not in 3NF because:
 - $\text{BookTitle} \rightarrow \text{BookType} \rightarrow \text{ListPrice}$
- BookType is neither a key itself nor a subset of a key and ListPrice is not a prime attribute
- The 3NF decomposition will eliminate the transitive dependency of Listprice. 3NF decomposition:
 - Book1(BookTitle, AuthorName)
 - Book2A(BookTitle, BookType, Publisher)
 - Book2B(BookType, ListPrice)
 - Book3(AuthorName, AuthorAffiliation)

Book1

<u>BookTitle</u>	<u>AuthorName</u>
------------------	-------------------

Book2A

<u>BookTitle</u>	<u>BookType</u>	<u>Publisher</u>
------------------	-----------------	------------------

Book2B

<u>BookType</u>	<u>ListPrice</u>
-----------------	------------------

Book3

<u>AuthorName</u>	<u>AuthorAffiliation</u>
-------------------	--------------------------

Problem 2

Consider the following relation:

CAR_SALE(Car#, DateSold, Salesman#, Commission%,
DiscountAmount)

Assume that a car may be sold by multiple salesmen, and hence
 $\{Car\#, Salesman\#\}$ is the primary key.

Additional dependencies are:

$Car\# \rightarrow DateSold$

$Car\# \rightarrow DiscountAmount$

$DateSold \rightarrow DiscountAmount$

$Salesman\# \rightarrow Commission\%$

Based on the given primary key, is the relation in 1NF, 2NF, 3NF?

Why or why not?

How would you successively normalize it completely?

Solution:

The relation is in 1NF because all attribute values are single atomic values.

- The relation is not in 2NF because:

- $\text{Car\#} \rightarrow \text{DateSold}$
- $\text{Car\#} \rightarrow \text{DiscountAmount}$
- $\text{Salesman\#} \rightarrow \text{Commission\%}$

Thus, these attributes are not fully functionally dependent on the primary key.

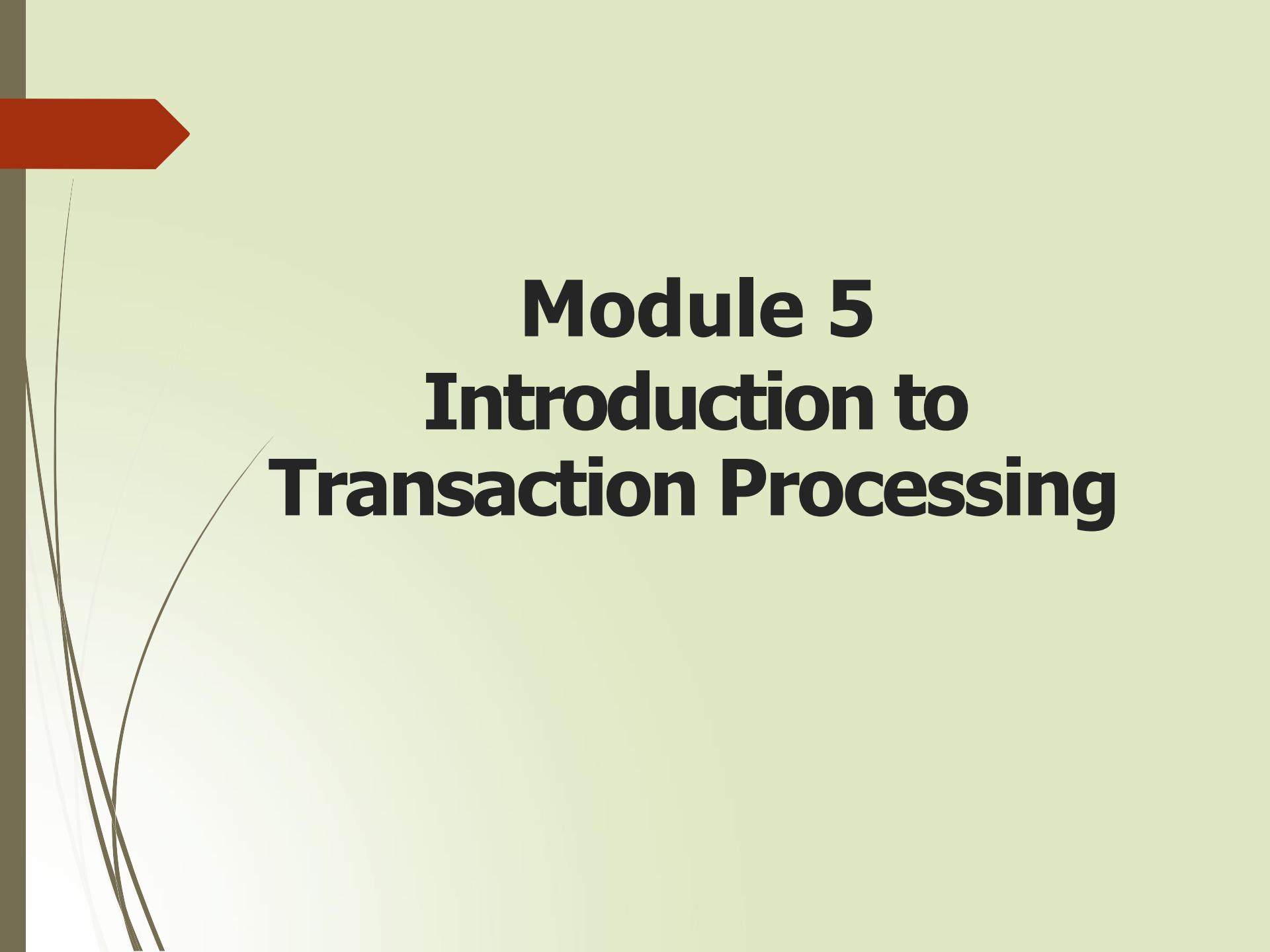
2NF decomposition:

- CAR_SALE1(Car#, DateSold, DiscountAmount)
 - CAR_SALE2(Car#, Salesman#)
 - CAR_SALE3(Salesman#, Commission%)
-
- The relations are not in 3NF because:
 - $\text{Car\#} \rightarrow \text{DateSold} \rightarrow \text{DiscountAmount}$
- DateSold is neither a key itself nor a subset of a key and
DiscountAmount is not a prime attribute.

- 3NF decomposition:

- CAR_SALES1A(Car#, DateSold)

- | CAR_SALES1B(DateSold, DiscountAmount)
 - | CAR_SALE2(Car#, Salesman#)
 - | CAR_SALE3(Salesman#, Commission%)



Module 5

Introduction to

Transaction Processing

Classifying database system: Single user Vs Multiuser system

At most one user is using the DB. Eg: Personal Computer System

- Many users are concurrently accessing the database Eg: Airline reservation database.
- Concurrent access is possible because of Multiprogramming.
- Multiprogramming can be achieved by:
 - interleaved execution
 - Parallel Processing

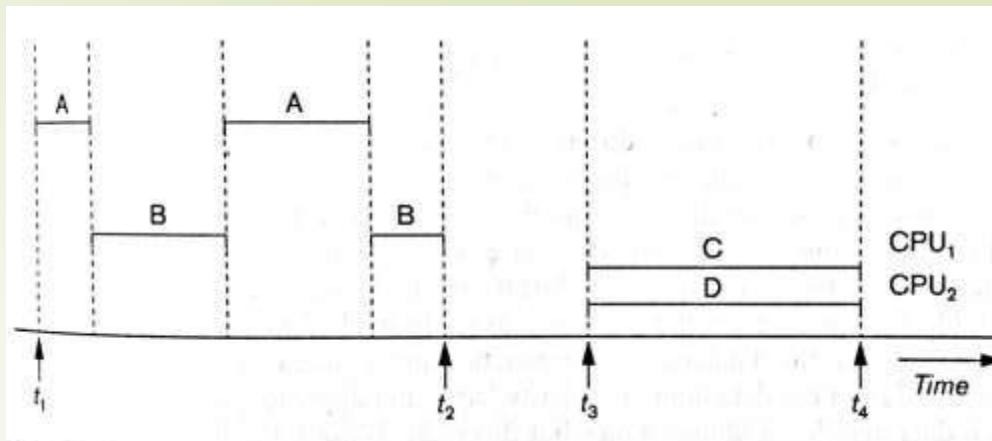


Figure 20.1
Interleaved
processing versus
parallel processing
of concurrent
transactions.

Operations and DBMS Buffers

- A Transaction is an executing program that forms a logical unit of database processing.
- It includes one or more DB access operations such as insertion, deletion, modification or retrieval operation.
- It can be either embedded within an application program using **begin transaction** and **end transaction** statements
- Or specified interactively via a high level query language such as SQL.
- Transaction which do not update database are known as **read only transactions**.
- Transaction which do update database are known as **read write transactions**.
- Items in the DB are organized into various levels of **granularity** such as: attribute, table, record, disk block etc. Each item is identified by a unique name.

Basic DB access operations that a transaction can include are:

`read_item(X)`: Reads a DB item named X into a program variable.

`write_item(X)`: Writes the value of a program variable into the DB item named X.

Executing `read_item(X)` include the following steps:

1. Find the block containing X
2. Copy the block into a buffer in main memory
3. Copy X from block to program variable X.

Executing `write_item(X)` include the following steps:

1. Find the block containing X
2. Copy the block into a buffer in main memory
3. Copy X from program variable named X into its correct location in buffer.

- Decision of when to store a modified disk block is handled by recovery manager of the DBMS in cooperation with operating system.
- A DB cache includes a number of data buffers.
- When the buffers are all occupied a buffer replacement policy is used to choose one of the buffers to be replaced. EG: LRU
- A transaction includes read_item and write_item operations to access and update DB.
- Read_set of T₁={X,Y}
- Write_set of T₁ ={X,Y}

Figure 20.2

Two sample transactions.

- (a) Transaction T₁.
(b) Transaction T₂.

(a)

T ₁
read_item(X); $X := X - N;$ write_item(X); read_item(Y); $Y := Y + N;$ write_item(Y);

(b)

T ₂
read_item(X); $X := X + M;$ write_item(X);

Why concurrency control is needed

- We consider an Airline reservation DB
- Each record is stored for an airline flight which includes Number of reserved seats among other information.

Figure 20.2

Two sample transactions.

- (a) Transaction T_1 .
- (b) Transaction T_2 .

(a)

T_1

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

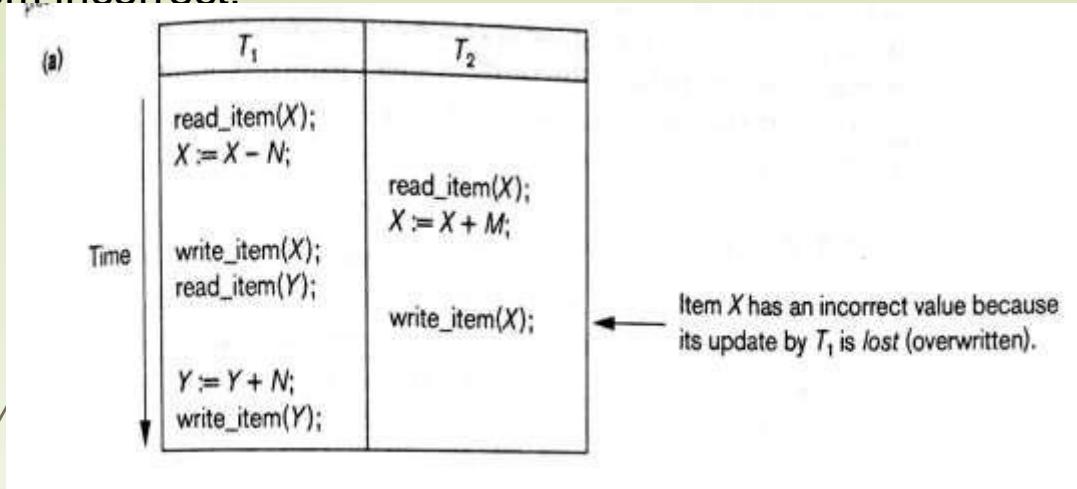
(b)

T_2

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

Lost Update Problem

This problem occurs when two transactions that access the same DB items have their operations interleaved in a way that makes the value of some DB item incorrect.



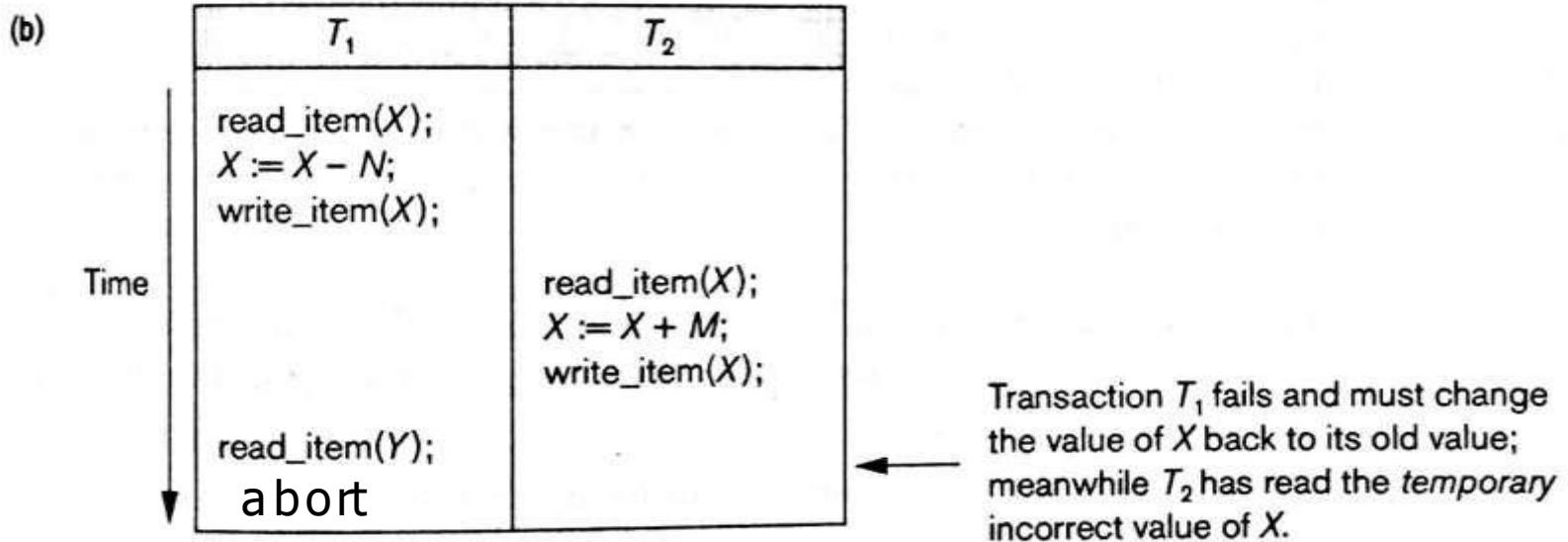
Eg: If $X=80$ (Reservations available on flight)

$N=5$ (No of reservations transferred from X to Y)

$M=4$ (Reservation on X)

Temporary Update(Dirty Read Problem)

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



Incorrect Summary Problem

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

(c)

T_1	T_3
<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>sum := 0; read_item(A); sum := sum + A; : read_item(X); sum := sum + X; read_item(Y); sum := sum + Y;</pre> <p>$\leftarrow T_3$ reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).</p>

Unrepeatable read Problem

Transaction T reads the same item twice and gets different values on each read, since the item was modified by another transaction T' between the two reads.

Why recovery is needed?

- Either all operations of a transaction are successful and their effect updated on the DB or transaction does not have any effect on the DB
- Committed transactions
- Aborted transactions

Types of failures:

1. Computer Failure/System crash: Hardware, software or system crash
2. Transaction/System error: Eg: integer overflow or division by zero
3. Local error or exception: errors in a transaction such as data not available.
4. Concurrency control enforcement: Aborts transaction since it violates serializability.
5. Disk failure:
6. Physical problems and catastrophes: general problems.

Transaction and System Concepts

Transaction states and additional operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all

For recovery the system keeps track of start of a transaction, termination, commit or aborts.

- BEGIN_TRANSACTION

- READ or WRITE

- END_TRANSACTION

COMMIT_TRANSACTION

- ROLLBACK

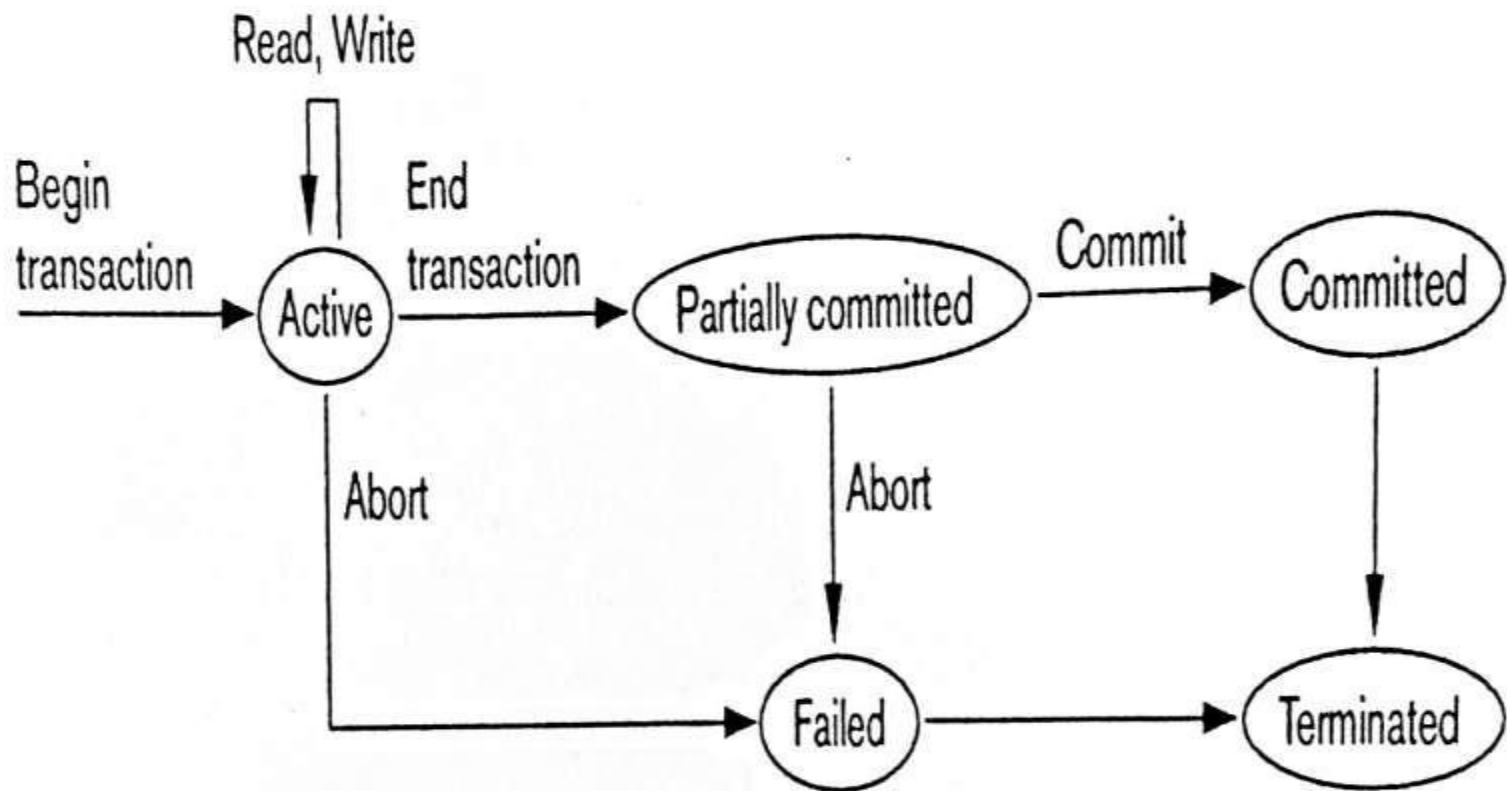


Figure 20.4

State transition diagram illustrating the states for transaction execution.

A transaction goes into **active state** immediately after it starts execution and can execute read and write operations.

When the transaction ends it moves to **partially committed state**.

At this end additional checks are done to see if the transaction can be committed or not. If these checks are successful the transaction is said to have reached commit point and enters **committed state**. All the changes are recorded permanently in the db.

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 operation.

Terminated state corresponds to the transaction leaving the system. All the information about the transaction is removed from system tables.

The system Log

For every transaction a unique transaction-id is generated by the system.

[start_transaction, transaction-id]: the start of execution of the transaction identified by transaction-id

[read_item, transaction-id, X]: the transaction identified by transaction-id reads the value of database item X.

[write_item, transaction-id, X, old_value, new_value]: the transaction identified by transaction-id changes the value of database item X from old_value to new_value

[commit, transaction-id]: the transaction identified by transaction-id has completed all accesses to the database successfully and its effect can be recorded permanently (committed)

[abort, transaction-id]: the transaction identified by transaction-id has been aborted

DBMS specific buffer Replacement policies

- **Domain Separation(DS) method:** In a DBMS, various types of disk pages exist: **index page, data file page, log file page** and so on.
- DBMS cache is divided into separate domains, each handles one type of disk pages and replacements within each domain are handled via basic LRU page replacement.
- LRU is a **static** algorithm and does not adopt to dynamically changing loads because the number of available buffers for each domain is predetermined.
- **Group LRU** adds dynamically load balancing feature since it gives each domain a priority and selects pages from lower priority level domain first for replacement.
- **Hot Set Method:** This is useful in queries that have to scan a set of pages repeatedly. The hot set method determines for each db processing algorithm the set of disk pages that will be accessed repeatedly and it does not replace them until their processing is completed.

The DBMIN method: uses a model known as QLSM (Query Locality set model), which predetermines the pattern of page references for each algorithm for a particular db operation. Depending on the type of access method, the file characteristics, and the algorithm used the QLSM will estimate the number of main memory buffers needed for each file involved in the operation.

Desirable Properties of a Transactions

- A **Atomicity:** a transaction is an atomic unit of processing and it is either performed entirely or not at all.
- C **Consistency Preservation:** a transaction should be consistency preserving that is it must take the database from one consistent state to another.
- I **Isolation/Independence:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executed concurrently.
- D **Durability (or Permanency):** if a transaction changes the database and is committed, the changes must never be lost because of any failure.

Atomicity is maintained by recovery subsystem

Consistency is the responsibility of programmer

Isolation is enforced by the concurrency control subsystem

Durability is the responsibility of recovery subsystem.

Characterizing Schedule based on recoverability

When transactions are executing concurrently in an interleaved fashion then the order of execution of operations from all the various transactions is known as a schedule (or History).

- A schedule S of n transactions T_1, T_2, \dots, T_n is a sequential ordering of the operations of the n transactions.
 - The transactions are interleaved
- A schedule maintains the order of operations within the individual transaction.
 - For each transaction T if operation a is performed in T before operation b, then operation a will be performed before operation b in S.
 - The operations are in the same order as they were before the transactions were interleaved- **Total Ordering**
- Two operations conflict if they belong to different transactions, AND access the same data item AND one of them is a write.
- Changing the order of the conflicting operations changes the end result.

The diagram illustrates two parallel processes, T_1 and T_2 , running over time. A vertical arrow on the left indicates the progression of time from top to bottom. Process T_1 consists of the following sequence of operations:

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

Process T_2 consists of the following sequence of operations:

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

$S_a: r_1(X); r_2(X); w_1(X) ; r_1(Y); w_2(X); w_1(Y);$

Conflicting operations:

- | | |
|---|---------------------|
| $r_1(X)$ conflicts with $w_2(X)$ | Read write conflict |
| $r_2(X)$ conflicts with $w_1(X)$ | |
| $w_1(X)$ conflicts with $w_2(X)$ | |
| $r_1(X)$ do not conflicts with $r_2(X)$ | |

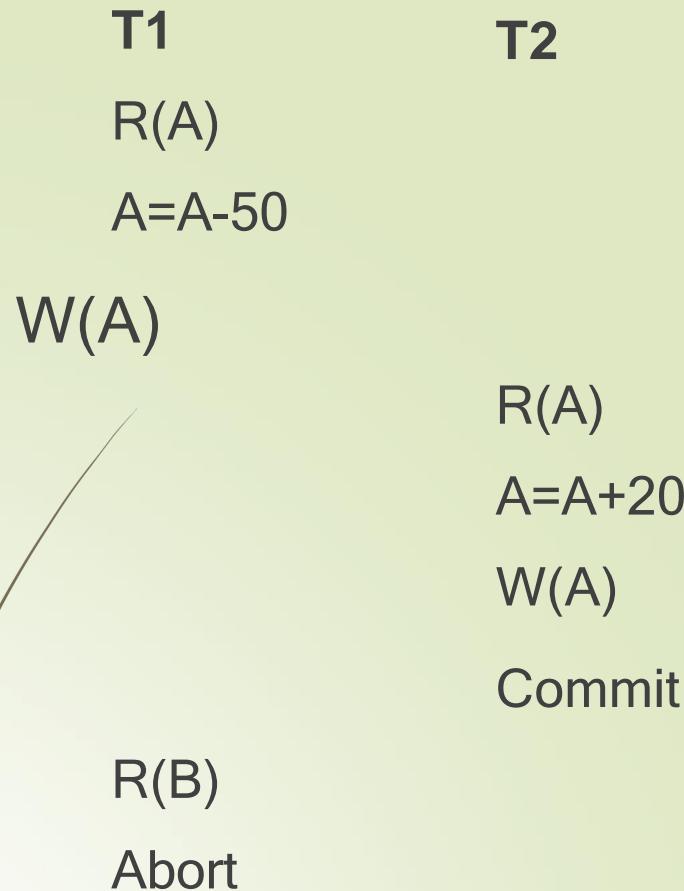
Complete schedule:

A schedule S in n transactions is said to be complete if the following conditions hold:

1. Each transaction in a schedule includes either a **commit** or **abort** as its last operation.
2. Relative order of each pair of operations in a transaction is preserved in S.
3. Non conflicting operations can occur in any order.

A complete schedule will not have any incomplete transaction at the end of the schedule.

A schedule is **recoverable** if no transaction T in S commits until all transaction T` that have written some item X that T reads have committed. Otherwise the schedule is non recoverable.



Example for Non Recoverable Schedule

In a recoverable schedule no committed transaction ever needs to be

Strict Schedule: In which a transaction can neither read or write an item X until the last transaction that wrote X has committed. Undo operation in a strict schedule is simply to restore the before image of data item X.

Characterizing Schedule based on Serializability

Let a schedule contain two transactions T1 and T2 accessing an airline DB, two serial schedules are possible:

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

(a)

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

Schedule A

(b)

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

Schedule B

(c)

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

Time ↓

Schedule C

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

Time ↓

Schedule D

A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions. There are $n!$ possible serial schedules.

Conflict Serializable: Two schedules are said to be conflict equivalent if the relative order of any two conflicting operations is the same in both schedules.

Serilizable schedule: A schedule S is serilizable if it is equivalent to some serial schedule.

(a)

T_1	T_2
Time ↓ <pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>read_item(X); X := X + M; write_item(X);</pre>

Schedule A

(c)

T_1	T_2
Time ↓ <pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>read_item(X); X := X + M; write_item(X);</pre>

Schedule C

Serializable schedule

Testing for serializability of a schedule using precedence graph:

Which is a directed graph $G=(N,E)$ that consist of a set of nodes

$N=\{T_1, T_2, \dots, T_n\}$ and a set of directed edges

$E=\{e_1, e_2, \dots, e_m\}$.

Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$,
 $1 \leq j \leq n, 1 \leq k \leq n$.

Such an edge from node T_j to T_k is created by the algorithm if a pair of conflicting operations exist in T_j and T_k and the operation appears in T_j before it appears in T_k .

Algorithm 20.1. 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 $\text{read_item}(X)$ after T_i executes a $\text{write_item}(X)$, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a $\text{write_item}(X)$ after T_i executes a $\text{read_item}(X)$, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a $\text{write_item}(X)$ after T_i executes a $\text{write_item}(X)$, 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.

(a)

T_1	T_2
<code>read_item(X); $X := X - N;$ <code>write_item(X); <code>read_item(Y); $Y := Y + N;$ <code>write_item(Y);</code></code></code></code>	

Time

Schedule A

(b)

T_1	T_2
<code>read_item(X); $X := X + M;$ <code>write_item(X); <code>read_item(Y); $Y := Y + N;$ <code>write_item(Y);</code></code></code></code>	

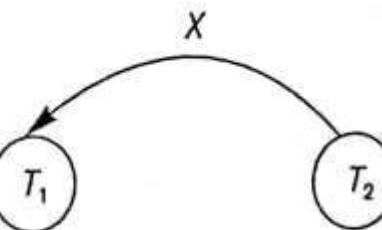
Time

Schedule B

(a)



(b)



(c)

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

Time

Schedule C

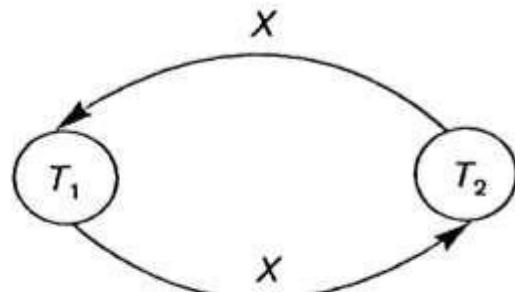
(d)

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

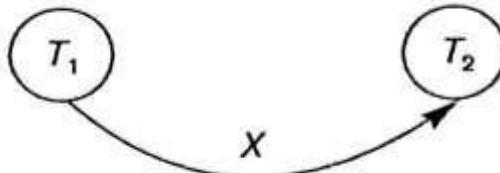
Time

Schedule D

(c)



(d)



(a)

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

(b)

Transaction T_1	Transaction T_2	Transaction T_3
Time ↓	read_item(Z); read_item(Y); write_item(Y);	read_item(Y); read_item(Z); write_item(Y); write_item(Z);
	read_item(X); write_item(X);	write_item(X);

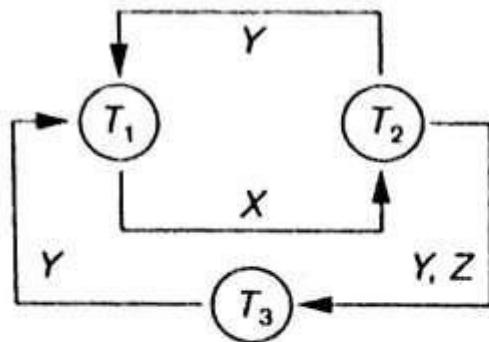
Schedule E

(c)

Transaction T_1	Transaction T_2	Transaction T_3
Time ↓	read_item(X); write_item(X);	read_item(Y); read_item(Z); write_item(Y); write_item(Z);
	read_item(Y); write_item(Y);	read_item(Z); read_item(Y); write_item(Y); read_item(X); write_item(X);

Schedule F

(d)



Equivalent serial schedules

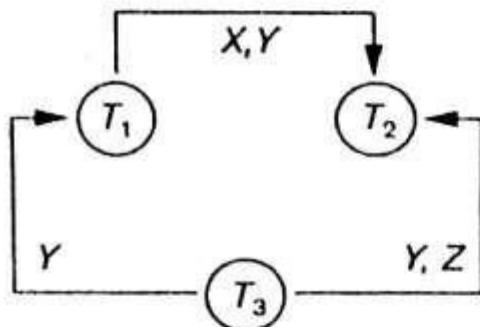
None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

Cycle $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

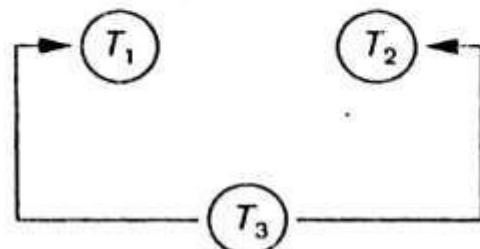
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

View Equivalence: Two schedules S and S' are View Equivalent if the following three conditions hold.

- As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.
- The same set of transactions participate in S and S' .
- The final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules
- A schedule S is view serializable if it is view equivalent to a serial schedule.
- Testing for view serializability is NP-complete

then one or more of the following violations may occur:

Dirty read

Nonrepeatable Read

Phantoms

Table 20.1 Possible Violations Based on Isolation Levels as Defined in SQL

Isolation Level	Type of Violation		
	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

Concurrency control techniques

The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.

- A lock is a variable associated with a data item in the database.
Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item. It is used for synchronizing the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

Binary locks have two possible states:

1. locked (`lock_item(X)` operation) and
2. unlocked (`unlock_item(X)` operation)

`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;

and control access to locks.

Shared/ Exclusive/ Multiple-mode (or Read/Write) locks

Allow concurrent access to the same item by several transactions. Three possible states:

1. read locked or shared locked (other transactions are allowed to read the item)
2. write locked or exclusive locked (a single transaction exclusively holds the lock on the item) and
3. unlocked.

One method to implement operations on read/write lock is to keep track of the number of transactions that hold a shared lock on an item in the lock table.

Each record in the lock table have four fields:

< Data item, LOCK, No_of_reads, locking transaction(s)>

Locking and Unlocking Operations for Shared/exclusive locks

read_lock(X):

B: if $\text{LOCK}(X) = \text{"unlocked"}$

then begin $\text{LOCK}(X) \leftarrow \text{"read-locked"};$

$\text{no_of_reads}(X) \leftarrow 1$

end

else if $\text{LOCK}(X) = \text{"read-locked"}$

then $\text{no_of_reads}(X) \leftarrow \text{no_of_reads}(X) + 1$

else begin

wait (until $\text{LOCK}(X) = \text{"unlocked"}$

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 $\text{LOCK}(X)$ = “write-locked”

then begin $\text{LOCK}(X) \leftarrow$ “unlocked”;

wakeup one of the waiting transactions, if any

end

else if $\text{LOCK}(X)$ = “read-locked”

then begin

$\text{no_of_reads}(X) \leftarrow \text{no_of_reads}(X) - 1$;

if $\text{no_of_reads}(X) = 0$

then begin $\text{LOCK}(X) =$ “unlocked”;

wakeup one of the waiting transactions, if any

end

end;

Conversion (Upgrading, Downgrading) of Locks

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

Eg: Transaction T that holds a `read_lock(X)` can then issue a `write_lock(X)` to upgrade a lock.

If T is the only transaction holding the lock this request will be granted. Otherwise the transaction must wait for others to release the lock.

It is also possible for a transaction T currently holding a `write_lock(X)` to issue a downgrading request with `read_item(X)`. Which will be immediately permitted.

serializability

X=20, Y=30

T1	T2
read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);	read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);

Schedule 1: T1 followed by T2 $\Rightarrow X=50, Y=80$

Schedule 2: T2 followed by T1 $\Rightarrow X=70, Y=50$

X=20
Y=30

T1	T2
<pre>read_lock(Y); read_item(Y); unlock(Y);</pre> <pre>write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);</pre>	<pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);</pre>

result of S \Rightarrow X=50, Y=50

Ensuring Serialisability: Two-Phase Locking -Basic 2PL

All locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.

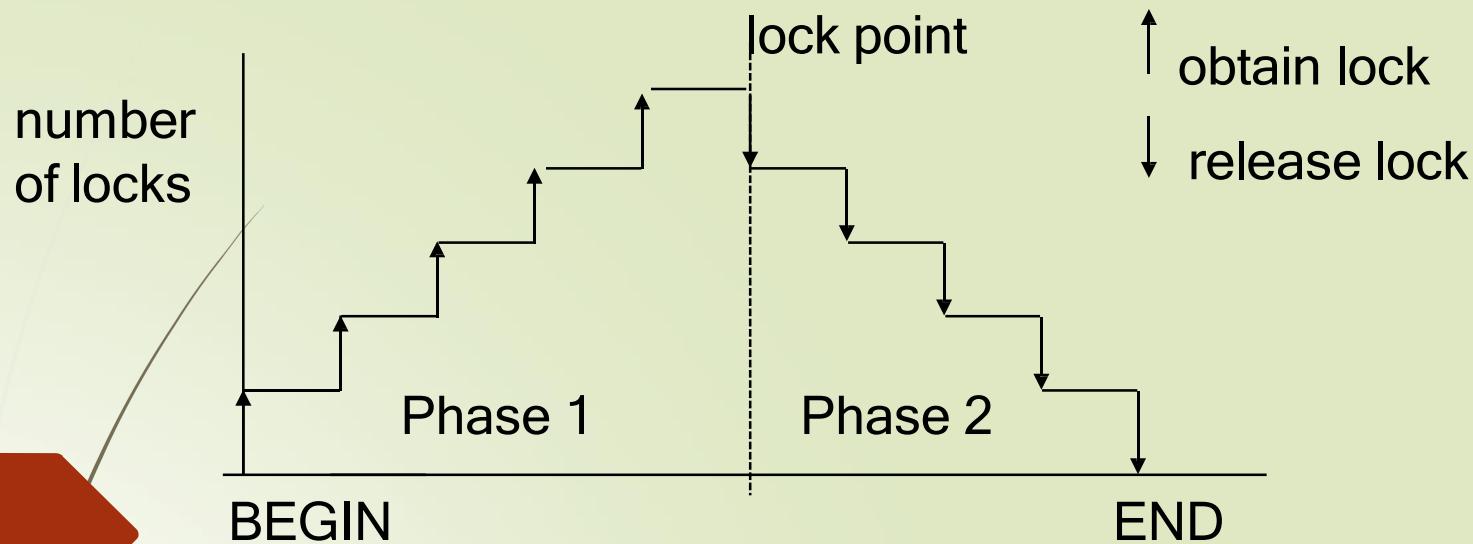
Two phases:

- || ***expanding phase***: new locks on items can be acquired but none can be released
- || ***shrinking phase***: existing locks can be released but no new ones can be acquired

X=20, Y=30

T1	T2
read_lock(Y); read_item(Y); write_lock(X); unlock(Y); read_item(X); X:=X+Y; write_item(X); unlock(X);	read_lock(X); read_item(X); write_lock(Y); unlock(X); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);

When a transaction releases a lock, it may not request another lock



Two phase locking guarantees serializability, but limits the amount of concurrency that can occur in a schedule

Locking Problems in Basic 2PL

- Each of two or more transactions is waiting for the other to release an item. This can create a **deadlock**.

T1	T2
<pre>read_lock(Y); read_item(Y); write_lock(X); Wait</pre>	<pre>read_lock(X); read_item(X); write_lock(Y); Wait</pre>

Variations of 2PL:

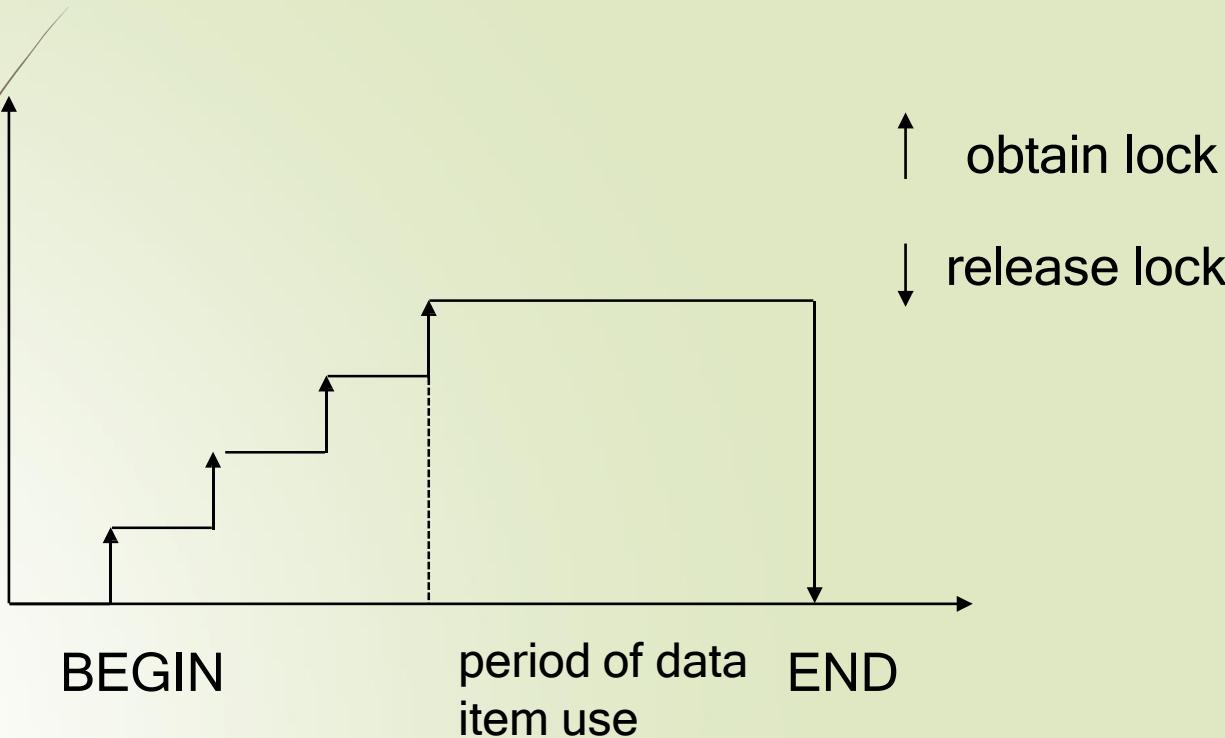
Conservative 2PL or static 2PL a transaction locks all the items it accesses before the transaction begins execution pre-declaring read and write sets. If any of the pre declared item cannot be locked, the transaction does not lock any item ; instead it waits for all required items to become available. It is deadlock free protocol

T1	T2
read_lock(Y); read_item(Y); write_lock(X);	read_lock(X); read_item(X); write_lock(Y);

Strict 2PL

A transaction does not release any of its exclusive locks until after it commits or aborts

- leads to a strict schedule for recovery
- It is also not deadlock free



Rigorous 2PL

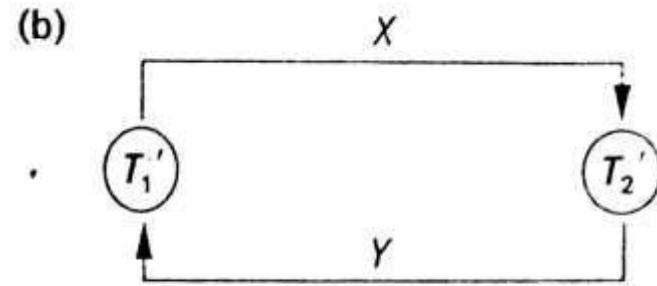
- A transaction does not release any of its locks until after it commits or aborts

Dealing with deadlock and starvation

(a)

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

Time ↓



Deadlock prevention protocol:

1) Transaction Timestamp

Each transaction is assigned a unique timestamp $TS(T')$ - a unique identifier assigned to each transaction

Timestamps are assigned in the order in which transactions are started;
hence if T_1 started before T_2 , then $TS(T_1) < TS(T_2)$

Two schemes to prevent deadlock are

Wait-die: If $TS(T_i) < TS(T_j)$, then (T_i is older than T_j) T_i is allowed to wait; otherwise (T_i is younger than T_j) abort T_i and restart it later with the same time stamp

Wound-wait:

If $TS(T_i) < TS(T_j)$, then (T_i is older than T_j) abort T_j (T_i wounds T_j) and restart it later with the same time stamp; otherwise (T_i is younger than T_j) T_i is allowed to wait.

Both schemes will abort a younger transaction, assuming that this will waste less processing. Both schemes are deadlock free.

However both schemes may cause some transactions to be aborted and restarted even though those transactions may never cause deadlock.

2) No waiting algorithm (NW)- If unable to obtain lock; abort and restart

3) Caution waiting algorithm (CW)- Suppose if T_i tries to lock an item X that is currently locked by T_j with a conflicting lock, then if T_j is not blocked (not waiting for some other item), then T_i is blocked and allowed to wait; otherwise abort T_i .

No transaction will ever wait for another blocked transaction.

Deadlock Detection:

1) Wait for graph:

- One node is created for each active transaction.
- whenever 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 on the item T_i was waiting for, the directed edge is dropped from the wait for graph.

- A deadlock is detected if the graph has a cycle.

2) Timeouts

Starvation:

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Fair waiting

Priority based on waiting time

Concurrency control based on timestamp ordering

Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp $\text{TS}(T_i)$, a new transaction T_j is assigned time-stamp $\text{TS}(T_j)$ such that $\text{TS}(T_i) < \text{TS}(T_j)$.

The protocol manages concurrent execution such that the time-stamps determine the serializability order.

In order to assure such behavior, the protocol maintains for each data X two timestamp values:

W-timestamp(X) is the largest time-stamp (T is the youngest transaction) of any transaction that executed **write(X)** successfully.

R-timestamp(X) is the largest time-stamp of any transaction that executed **read(X)** successfully.

If $\text{TS}(T_i) \leq \text{W-timestamp}(X)$ (Value modified by a younger transaction), then T_i needs to read a value of X that was already overwritten.

Hence, the **read** operation is rejected, and T_i is rolled back.

If $\text{TS}(T_i) \geq \text{W-timestamp}(X)$, then the **read** operation is executed, and $\text{R-timestamp}(X)$ is set to **max**($\text{R-timestamp}(X)$, $\text{TS}(T_i)$).

Suppose that transaction T_i issues **write(X)**.

1. If $TS(T_i) < R\text{-timestamp}(X)$ or $TS(T_i) < W\text{-timestamp}(X)$, then **write** operation is rejected and T_i is rolled back.
Since some younger transaction has already read or written X before T had a chance to write X.
2. Otherwise, the **write** operation is executed, and $W\text{-timestamp}(X)$ is set to $TS(T_i)$.
3. Whenever the basic TO algorithm detects two conflicting operations that occur in the incorrect order, it rejects the later of the two operation and aborts the transaction. The schedule produced here is guaranteed to be serializable.
4. But this can lead to cyclic restart and starvation.

Strict Timestamp Ordering

1. Transaction T issues a `write_item(X)` operation:
 - If $TS(T) > read_TS(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

2. Transaction T issues a `read_item(X)` operation:
 - If $TS(T) > write_TS(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- **Thomas's Write Rule**
 - If $\text{read_TS}(X) > \text{TS}(T)$ then abort and roll-back T and reject the operation.
 - If $\text{write_TS}(X) > \text{TS}(T)$, then just ignore the write operation and continue execution. This is because it is already outdated and obsolete.
 - If the conditions given in 1 and 2 above do not occur, then execute $\text{write_item}(X)$ of T and set $\text{write_TS}(X)$ to $\text{TS}(T)$.

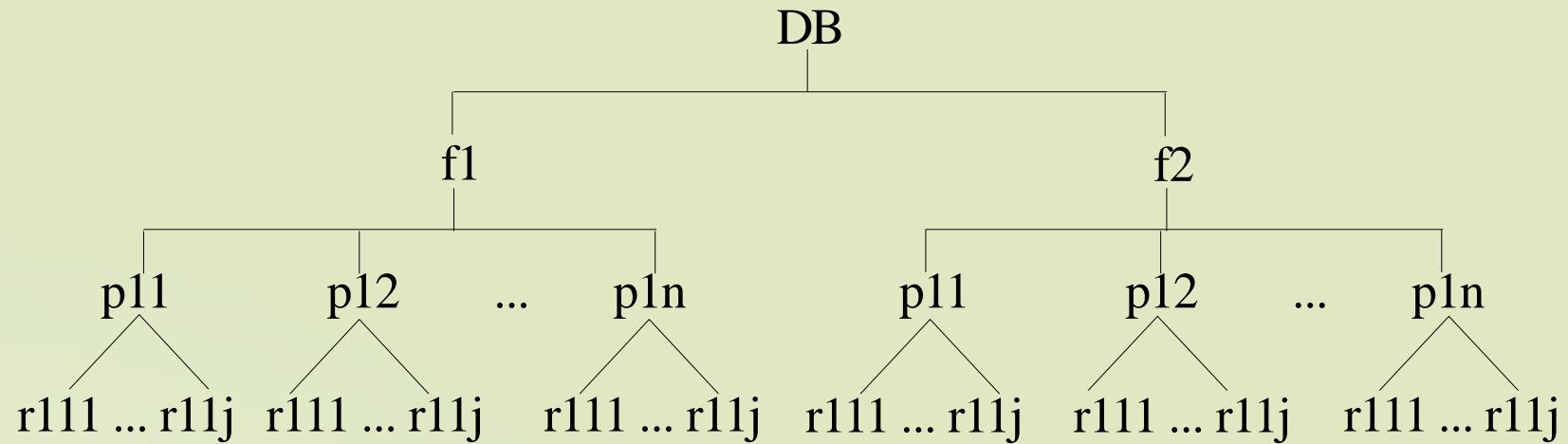
Multiversion technique based on timestamp

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction.
- Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effects: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

- || Assume X_1, X_2, \dots, X_n are the versions of a data item X created by write operations of transactions. With each X_i a **read_TS** (read timestamp) and a **write_TS** (write timestamp) are associated.
- || **read_TS(X_i)**: The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i .
- || **write_TS(X_i)**: The write timestamp of X_i that wrote the value of version X_i .
- || A new version of X_i is created only by a write operation.

Granularity of data items and Multiple Granularity Locking

- A block (a unit of data at a size of data item) defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation).
- Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity.
- Example of data item granularity:
 1. A field of a database record (an attribute of a tuple)
 2. A database record (a tuple or a relation)
 3. A disk block
 4. An entire file
 5. The entire database
- The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:

Intention-shared (IS): indicates that a shared lock(s) will be requested on some descendent nodes(s).

Intention-exclusive (IX): indicates that an exclusive lock(s) will be requested on some descendent node(s).

Shared-intention-exclusive (SIX): indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Intention-shared (IS)
Intention-exclusive (IX)
Shared-intention-exclusive (SIX)

Multiple Granularity Locking Protocol

The set of rules which must be followed for producing serializable schedule are

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

Rule 1 states that conflicting locks cannot be granted.

Rule 2,3 and 4 state the condition when a transaction may lock a given node in any lock modes.

Rule 5 and 6 enforce 2PL.

The process of locking begins from root node and goes down the tree until the node that needs to be locked is encountered.

Whereas unlocking starts from locked node and goes up the tree until the root itself is unlocked.

Consider three transactions:

T1 wants to update record r_{111} and record r_{211}

T2 wants to update all records on page p_{12} .

T3 wants to read record r_{11j} and the entire f_2 file.

An example of a serializable execution:

T1
IX(db)
IX(f1)

T2

T3

IX(p11)
X(r111)

IX(db)

IS(db)
IS(f1)
IS(p11)

IX(f1)
X(p12)

S(r11j)

IX(f2)
IX(p21)
X(r211)
Unlock (r211)
Unlock (p21)
Unlock (f2)

S(f2)

CONTD

unlock(r111)
unlock(p11)
unlock(f1)
unlock(db)

(r111j)

T1
unlock(p12)
unlock(f1)
unlock(db)

T2

unlock (p11)
unlock (f1)
unock(f2)

T3

unlock
unlock(db)

Database Recovery Techniques

Purpose of Database Recovery

- To bring the database into a consistent state after a failure occurs.
- To ensure the transaction properties of Atomicity (a transaction must be done in its entirety; otherwise, it has to be rolled back) and Durability (a committed transaction cannot be canceled and all its updates must be applied permanently to the database).
- After a failure, the DBMS recovery manager is responsible for bringing the system into a consistent state before transactions can resume.

Example:

If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

Recovery from non catastrophic transaction failure:

- **Deferred Update:** Do not physically update the db on the disk until the transaction commits, and update the recordes in the db.
- All modifications are maintained in the main memory buffer. Before reaching commit updates are recorded in the log file and then after commit, the updates are written to db from main memory buffer.
- If a transaction fails before reaching the commit point, it will not have changed the db on disk, so **no undo** is needed.
- It maybe necessary to REDO the effect of operations of a committed transaction from the log, because there effect would not have recorded on the db.
- Hence deferred update is also known as **NO-UNDO/REDO** algorithm.

- **Immediate Update:** DB maybe updated by **some operations** of a transaction before the transaction reaches its commit point. A data item modified in cache can be written to disk *before the transaction commits.*
- These operations must also be recorded in the log on disk by force writing before they are applied to the db on disk, making recovery still possible.
- If a transaction fails after updating the db, the effects of its operations must be rolled back from the db
- This technique is also known as **UNDO/REDO** algorithm, since both are required for recovery.
- A variation of this algorithm where **all updates** are required to be recorded in the db on disk before a transaction commit requires undo only, so it is known as **UNDO/NO-REDO** algorithm.

Caching of Disk blocks

Database Cache: A set of *main memory* buffers; each buffer typically holds contents of one disk block. Stores the disk blocks that contain the data items being read and written by the database transactions.

A **directory** for the cache is used to keep track of which db items are in buffer. This can be a table of

<Disk_page_address, Buffer_location, > entries.

When a page is first read from db to cache, a new entry is inserted into directory.

Each time a item is requested, DBMS first checks the cache directory to determine whether the page containing the item is in the DBMS cache.

The DBMS cache holds additional information relevant to buffer management. Each buffer has a **Dirty Bit**.

When the buffer contents are replaced from the cache, the contents must first be written back to the corresponding disk page only if the dirty bit is 1.

Pin-unpin bit: A page in the cache is pinned if it cannot be written back to disk as yet.

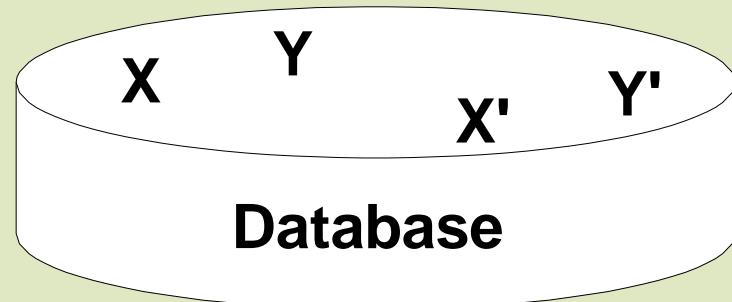
Eg: The recovery protocol may restrict certain buffer pages from being written back to the disk until the transaction that changed this buffer have committed.

A modified page can be flushed from buffer back to disk using two strategies :

In-place updating: Writes the buffer to the same disk location, thus overwriting the old value of updated item on disk. Hence a single copy of db disk block is maintained.

Shadowing: Writes an updated buffer to a **different location**, so multiple version of data items can be maintained.

Both the before image and after image can be kept on disks



Write-Ahead Logging, Steal/ No Steal and Force/ No Force:

The information needed for recovery must be written to the log file on disk before changes are made to the database on disk. This process is generally known as **Write-Ahead Logging (WAL)** protocol.

Two types of log entries are:

A Redo type log entry: Include the new value of the data item/AFIM, needed for Redo.

A Undo type log entry: Include the old value of the data item/BFIM, needed for Undo.

Steal/No-Steal and Force/No-Force

Specify how to flush database cache buffers to database on disk:

Steal: Cache buffers updated by a transaction may be flushed to disk before the transaction commits (recovery may require UNDO).

No-Steal: Cache buffers cannot be flushed until transaction commit (NO-UNDO). (Buffers are *pinned* till transactions commit).

Force: All pages updated by a transaction are immediately written(forced) to disk before transaction commits.

NO-REDO is required with Force approach.

No-Force: Some cache flushing may be delayed till transaction commits.

Recovery may require REDO.

These give rise to four different ways for handling recovery:

- Steal/No-Force (Undo/Redo),
- Steal/Force (Undo/No-redo),
- No-Steal/No-Force (Redo/No-undo),
- No-Steal/Force (No-undo/No-redo).

Checkpointing

Another type of entry in the log is called **checkpoint**, which is periodically written:

[checkpoint, list of Active transactions]

It minimizes the **REDO** operations during recovery.

The following steps define a checkpoint operation:

- Suspend execution of transactions temporarily.
- Force write modified buffers from cache to disk.
- Write a [checkpoint] record to the log, save the log to disk.

This record also includes other info., such as the *list of active transactions* at the time of checkpoint.

Resume normal transaction execution.

During recovery **redo** is required only for transactions that have committed *after the last [checkpoint] record* in the log.

A variation of checkpointing called **fuzzy checkpointing** allows transactions to continue execution during the checkpointing process.

Deferred Update (NO-UNDO/REDO) Recovery Protocol

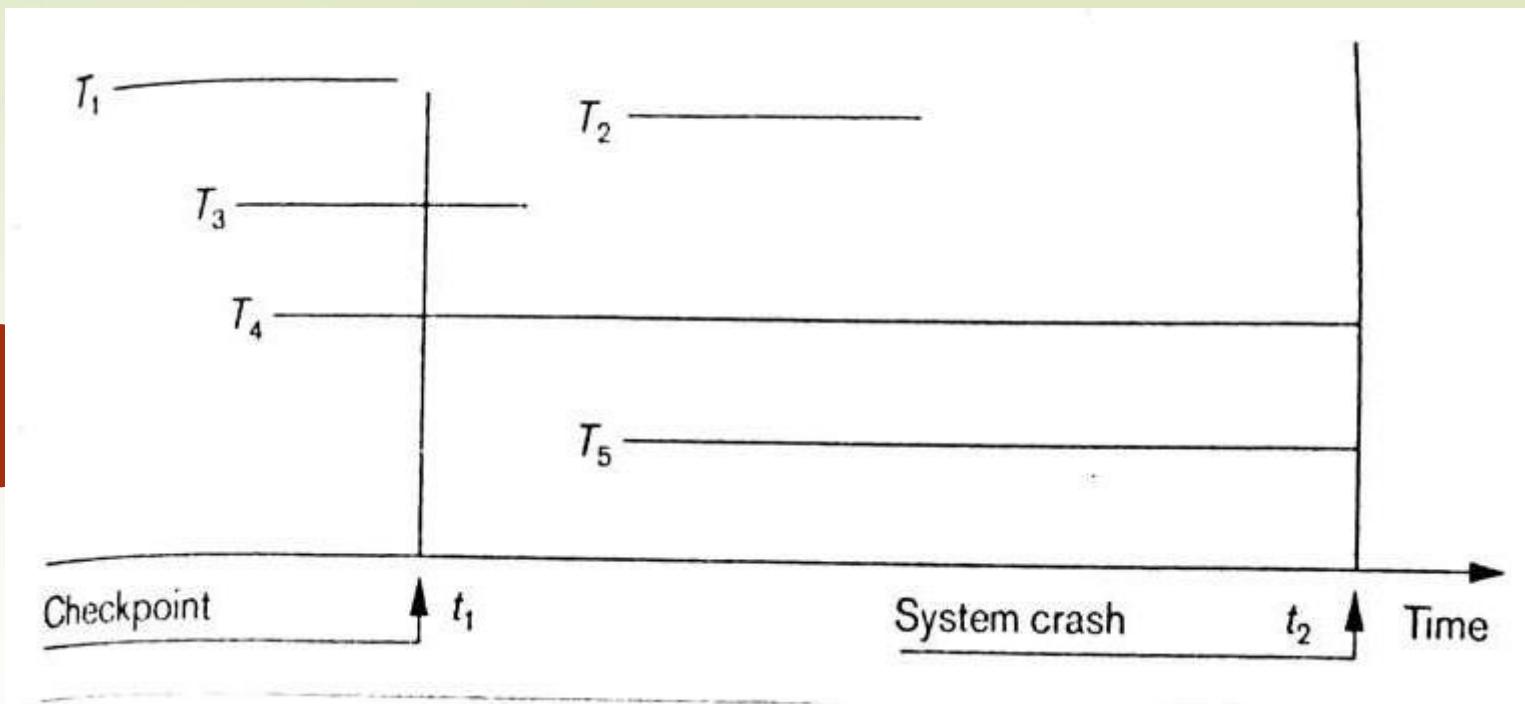
Recovery subsystem analyzes the log, and creates two lists:

Active Transaction list: All active (uncommitted) transaction ids are entered in this list.

Committed Transaction list: Transactions committed after the last checkpoint are entered in this table.

During recovery, transactions in **commit** list are **redone**; transactions in **active** list are *ignored* (because of **NO-STEAL** rule, none of their writes have been applied to the database on disk). Some transactions may be **redone** twice; this does not create inconsistency because **REDO** is “**idempotent**”, that is, one REDO for an AFIM is equivalent to multiple REDO for the same AFIM. (**RDU_M**-Recovery using deferred Update in a Multiuser environment)

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



I **Advantage:** Only **REDO** is needed during recovery.

Disadvantage: Many buffers may be pinned while waiting for transactions that updated them to commit, so system may run out of cache buffers when requests are made by new transactions. Limits concurrency

(a)

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(A)
write_item(A)
read_item(C)
write_item(C)

T_4
read_item(B)
write_item(B)
read_item(A)
write_item(A)

(b)

[start_transaction, T_1]
[write_item, T_1 , D , 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4 , B , 15]
[write_item, T_4 , A , 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2 , B , 12]
[start_transaction, T_3]
[write_item, T_3 , A , 30]
[write_item, T_2 , D , 25]

← System crash

 T_2 and T_3 are ignored because they did not reach their commit points. T_4 is redone because its commit point is after the last system checkpoint.**Figure 22.3**

An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions.

(b) System log at the point of crash.

Recovery techniques based on immediate Update

Only **UNDO-type log entries**: Includes only **BFIM** of the item.

It uses steal strategy.

Two main categories of immediate update algorithm:

UNDO/NO-REDO: using **steal / force**

UNDO/REDO: using **steal/no force**

Procedure RIU-M (Recovery using immediate updates for a multiuser environment):

1. Two lists maintained: committed transactions since the last checkpoint and active transaction.

Undo all write operation of active transaction , using UNDO procedure. Undo operation is performed in the reverse order from the log file.

3. Redo all write operation of committed transactions from the log, using the redo procedure.

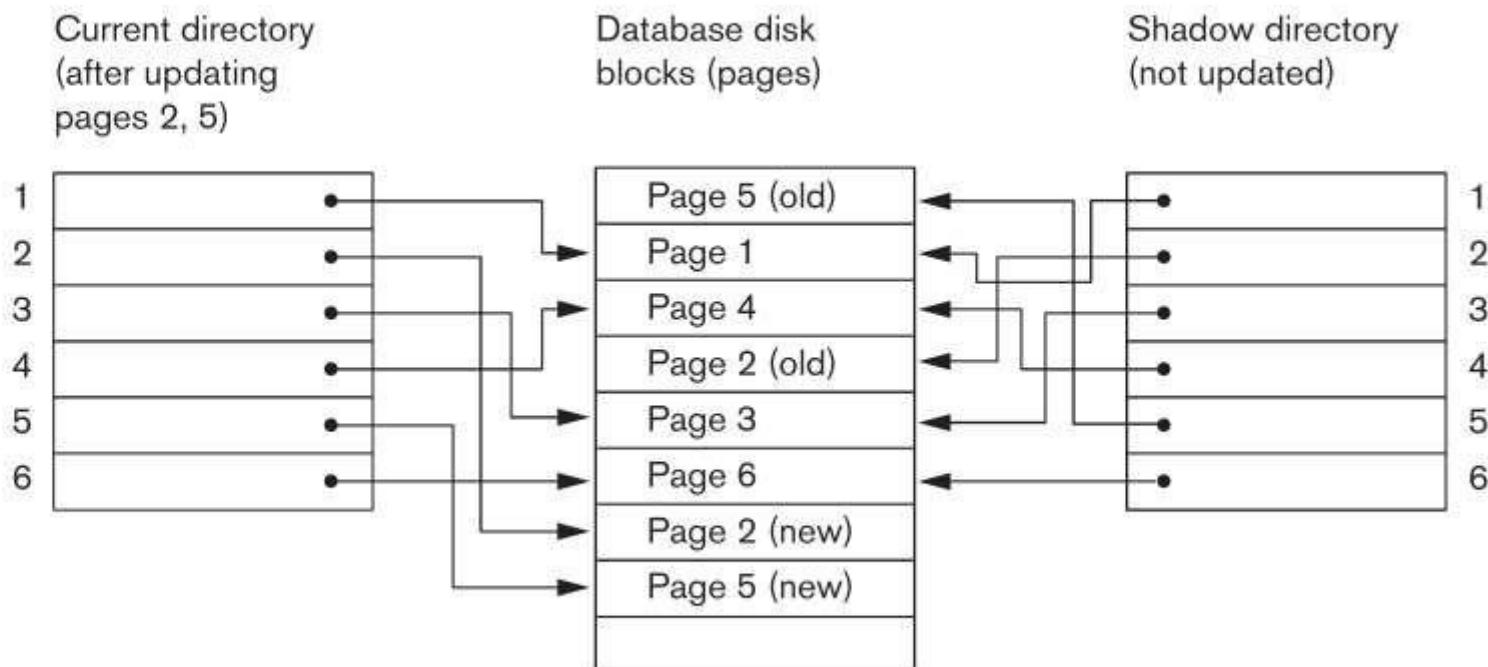
- **Procedure UNDO (WRITE_OP):** Undoing a write item operation WRITE_OP consist of examining its log entry [write_item, T, X, old_value, new_value] and setting the value of X in the db to old value, which is the after image.
- If multiple writes on X are performed then the earliest write is undone from the log.

- Does not require log file for single user environment.
- A directory with n entries is constructed where the ith entry points to the ith DB page on disk.
- When a transaction begins executing, the current directory- whose entries point to the most recent or current db pages on disk is copied into a **shadow directory**.
- During transaction execution the shadow directory is never modified.
- When a write_item operation is performed, a new copy of that page is not overwritten. Instead a new page is written elsewhere on unused disk block.

- The current directory entry is modified to point to new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.
- Old version is referenced by the shadow directory and new version by the current directory.

Figure 23.4

An example of shadow paging.



⁵The directory is similar to the page table maintained by the operating system for each process.

ARIES Recovery Algorithm

Used in practice, it is based on steal/NO force approach:

1. **WAL** (Write Ahead Logging)
2. **Repeating history during redo:** ARIES will retrace all actions of the database system prior to the crash to reconstruct the correct database state.
3. **Logging changes during undo:** It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery algorithm consists of three steps:

1. **Analysis:** step identifies the **dirty (updated) page buffers** in the cache and the set of **transactions active** at the time of crash. The set of transactions that committed after the last checkpoint is determined, and the appropriate point in the log where redo is to start is also determined.
2. **Redo:** Reapplies updates from the log to the db. Necessary redo operations are applied.
3. **Undo:** log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.

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

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

TRANSACTION TABLE

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

DIRTY PAGE TABLE

Page_id	Lsn
C	7
B	2
A	6

Figure 22.5

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