

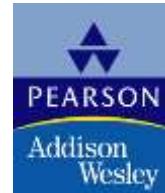
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 1

Introduction and Conceptual Modeling



Copyright © 2004 Pearson Education, Inc.

Types of Databases and Database Applications

- Numeric and Textual Databases
- Multimedia Databases
- Geographic Information Systems (GIS)
- Data Warehouses
- Real-time and Active Databases

A number of these databases and applications are described later in the book (see Chapters 24,28,29)

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.

Typical DBMS Functionality

- Define a database : in terms of data types, structures and constraints
- Construct or Load the Database on a secondary storage medium
- Manipulating the database : querying, generating reports, insertions, deletions and modifications to its content
- Concurrent Processing and Sharing by a set of users and programs – yet, keeping all data valid and consistent

Typical DBMS Functionality

Other features:

- Protection or Security measures to prevent unauthorized access
- “Active” processing to take internal actions on data
- Presentation and Visualization of data

Example of a Database (with a Conceptual Data Model)

- **Mini-world for the example:** Part of a UNIVERSITY environment.
- **Some mini-world *entities*:**
 - STUDENTs
 - COURSEs
 - SECTIONs (of COURSEs)
 - (academic) DEPARTMENTs
 - INSTRUCTORs

Note: The above could be expressed in the ENTITY-RELATIONSHIP data model.

Example of a Database (with a Conceptual Data Model)

- 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.

Main Characteristics of the Database Approach

- Self-describing 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.

Main Characteristics of the Database Approach

- 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.

Main Characteristics of the Database Approach

- Sharing of data and multiuser transaction processing : allowing a set of concurrent users to retrieve and to update the database. Concurrency control within the DBMS guarantees that each **transaction** is correctly executed or completely aborted. OLTP (Online Transaction Processing) is a major part of database applications.

Database Users

Users may be divided into those who actually use and control the content (called “Actors on the Scene”) and those who enable the database to be developed and the DBMS software to be designed and implemented (called “Workers Behind the Scene”).

Database Users

Actors on the scene

- **Database administrators:** responsible for authorizing access to the database, for co-ordinating and monitoring its use, acquiring software, and hardware resources, controlling its use and monitoring efficiency of operations.
- **Database Designers:** responsible to define the content, the structure, the constraints, and functions or transactions against the database. They must communicate with the end-users and understand their needs.
- **End-users:** they use the data for queries, reports and some of them actually update the database content.

Categories of End-users

- **Casual** : access database occasionally when needed
- **Naïve or Parametric** : they make up a large section of the end-user population. They use previously well-defined functions in the form of “canned transactions” against the database. Examples are bank-tellers or reservation clerks who do this activity for an entire shift of operations.

Categories of End-users

- **Sophisticated** : these include business analysts, scientists, engineers, others thoroughly familiar with the system capabilities. Many use tools in the form of software packages that work closely with the stored database.
- **Stand-alone** : mostly maintain personal databases using ready-to-use packaged applications. An example is a tax program user that creates his or her own internal database.

Advantages of Using the Database Approach

- Controlling redundancy in data storage and in development and maintenance efforts.
- Sharing of data among multiple users.
- Restricting unauthorized access to data.
- Providing persistent storage for program Objects (in Object-oriented DBMS's – see Chs. 20-22)
- Providing Storage Structures for efficient Query Processing

Advantages of Using the Database Approach

- Providing backup and recovery services.
- Providing multiple interfaces to different classes of users.
- Representing complex relationships among data.
- Enforcing integrity constraints on the database.
- Drawing Inferences and Actions using

Additional Implications of Using the Database Approach

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

Additional Implications of Using the Database Approach

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

Historical Development of Database Technology

- **Early Database Applications:** The Hierarchical and Network Models were introduced in mid 1960's and dominated during the seventies. A bulk of the worldwide database processing still occurs using these models.
- **Relational Model based Systems:** The model that was originally introduced in 1970 was heavily researched and experimented with in IBM and the universities. Relational DBMS Products emerged in the 1980's.

Historical Development of Database Technology

- **Object-oriented applications:** OODBMSs were introduced in late 1980's and early 1990's to cater to the need of complex data processing in CAD and other applications. Their use has not taken off much.
- **Data on the Web and E-commerce Applications:** Web contains data in HTML (Hypertext markup language) with links among pages. This has given rise to a new set of applications and E-commerce is using new standards like XML (eXtended Markup Language).

Extending Database Capabilities

- **New functionality is being added to DBMSs in the following areas:**
 - Scientific Applications
 - Image Storage and Management
 - Audio and Video data management
 - Data Mining
 - Spatial data management
 - Time Series and Historical Data Management

The above gives rise to new research and development in incorporating new data types, complex data structures, new operations and storage and indexing schemes in database systems.

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, concurrency control, recovery, and integrity functions.

- **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.

When not to use a DBMS

● 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.

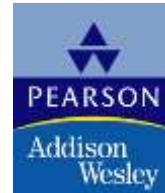
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 2

Database System Concepts and Architecture



Copyright © 2004 Pearson Education, Inc.

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.
Operations on the data model may include *basic operations* and *user-defined operations*.

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 (representational)** data models:
Provide concepts that fall between the above two, balancing user views with some computer storage details.

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 (DB2, ORACLE, SQL Server, SYBASE, INFORMIX).
- 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 Equipment Corp.).
- 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.)

History of Data Models

- 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. Started with Informix Universal Server. Exemplified in the latest versions of Oracle-10i, DB2, and SQL Server etc. systems.

Hierarchical Model

- **ADVANTAGES:**
 - Hierarchical Model is simple to construct and operate on
 - Corresponds to a number of natural hierarchically organized domains - e.g., assemblies in manufacturing, personnel organization in companies
 - Language is simple; uses constructs like GET, GET UNIQUE, GET NEXT, GET NEXT WITHIN PARENT etc.
- **DISADVANTAGES:**
 - Navigational and procedural nature of processing
 - Database is visualized as a linear arrangement of records
 - Little scope for "query optimization"

Network Model

- ADVANTAGES:
 - Network Model is able to model complex relationships and represents semantics of add/delete on the relationships.
 - Can handle most situations for modeling using record types and relationship types.
 - 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"

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.
- **Schema Construct:** A component of the schema or an object within the schema, e.g., STUDENT, COURSE.
- **Database Instance:** The actual data stored in a database at a *particular moment in time*. Also called **database state** (or **occurrence**).

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 when it is loaded
- **Valid State:** A state that satisfies the structure and constraints of the database.
- **Distinction**
 - 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**.

Three-Schema Architecture

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

Three-Schema Architecture

- Defines DBMS schemas at *three levels*:
 - **Internal schema** at the internal level to describe physical 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 for a community of users. 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.

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 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.

Data Independence

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

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.

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

DBMS Languages

- **High Level or Non-procedural Languages:** e.g., SQL, are *set-oriented* and specify what data to retrieve than how to retrieve. Also called *declarative* languages.
- **Low Level or Procedural Languages:** record-at-a-time; they specify *how* to retrieve data and include constructs such as looping.

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, 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

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 accounts, granting authorizations
 - Setting system parameters
 - Changing schemas or access path

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.

Other Tools

- **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.
- **Application Development Environments and CASE (computer-aided software engineering) tools:**
 - Examples – Power builder (Sybase), Builder (Borland)

Centralized and Client-Server Architectures

- **Centralized DBMS:** combines everything into single system including- DBMS software, hardware, application programs and user interface processing software.

Basic Client-Server Architectures

- **Specialized Servers with Specialized functions**
- **Clients**
- **DBMS Server**

Specialized Servers with Specialized functions:

- File Servers
- Printer Servers
- Web Servers
- E-mail Servers

Clients:

- Provide appropriate interfaces and a client-version of the system to access and utilize the server resources.
- Clients maybe 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
- Sometimes called query and transaction servers

Two Tier Client-Server Architecture

- **User Interface Programs and Application Programs** run on the client side
- Interface called **ODBC (Open Database Connectivity** – see Ch 9) provides an Application program interface (API) allow client side programs to call the DBMS. Most DBMS vendors provide ODBC drivers.

Two Tier Client-Server Architecture

- A client program may connect to several DBMSs.
- Other variations of clients are possible: e.g., in some DBMSs, more functionality is transferred to clients including data dictionary functions, optimization and recovery across multiple servers, etc. In such situations the server may be called the **Data Server**.

Three Tier Client-Server Architecture

- Common for **Web applications**
- Intermediate Layer called **Application Server or Web Server:**
 - stores the web connectivity software and **the rules and business logic (constraints)** part of the application used to access the right amount of data from the database server
 - acts like a conduit for sending partially processed data between the database server and the client.
- **Additional Features- Security:**
 - encrypt the data at the server before transmission
 - decrypt data at the client

Classification of DBMSs

- **Based on the data model used:**
 - Traditional: Relational, Network, Hierarchical.
 - Emerging: Object-oriented, Object-relational.
- **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)

Classification of DBMSs

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.*

Variations of Distributed Environments:

- **Homogeneous DDBMS**
- **Heterogeneous DDBMS**
- **Federated or Multidatabase Systems**

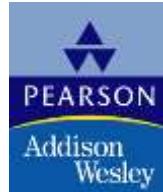
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 3

Data Modeling Using the Entity-Relationship (ER) Model



Chapter Outline

- Example Database Application (COMPANY)
- ER Model Concepts
 - Entities and Attributes
 - Entity Types, Value Sets, and Key Attributes
 - Relationships and Relationship Types
 - Weak Entity Types
 - Roles and Attributes in Relationship Types
- ER Diagrams - Notation
- ER Diagram for COMPANY Schema
- Alternative Notations – UML class diagrams, others

Example COMPANY Database

- Requirements of the Company (oversimplified for illustrative purposes)
 - The company is organized into DEPARTMENTS. Each department has a name, number and an employee who *manages* the department. We keep track of the start date of the department manager.
 - Each department *controls* a number of PROJECTS. Each project has a name, number and is located at a single location.

Example COMPANY Database (Cont.)

- We store each EMPLOYEE's social security number, address, salary, sex, and birthdate. Each employee *works for* one department but may *work on* several projects. We keep track of the number of hours per week that an employee currently works on each project. We also keep track of the *direct supervisor* of each employee.
- Each employee may *have* a number of DEPENDENTS. For each dependent, we keep track of their name, sex, birthdate, and relationship to employee.

ER Model Concepts

● Entities and Attributes

- Entities are specific objects or things in the mini-world that are represented in the database. For example the EMPLOYEE John Smith, the Research DEPARTMENT, the ProductX PROJECT
- Attributes are properties used to describe an entity. For example an EMPLOYEE entity may have a Name, SSN, Address, Sex, BirthDate
- A specific entity will have a value for each of its attributes. For example a specific employee entity may have Name='John Smith', SSN='123456789', Address ='731, Fondren, Houston, TX', Sex='M', BirthDate='09-JAN-55'
- Each attribute has a *value set* (or data type) associated with it – e.g. integer, string, subrange, enumerated type, ...

Types of Attributes (1)

- Simple
 - Each entity has a single atomic value for the attribute. For example, SSN or Sex.
- Composite
 - The attribute may be composed of several components. For example, Address (Apt#, House#, Street, City, State, ZipCode, Country) or Name (FirstName, MiddleName, LastName). Composition may form a hierarchy where some components are themselves composite.
- Multi-valued
 - An entity may have multiple values for that attribute. For example, Color of a CAR or PreviousDegrees of a STUDENT. Denoted as {Color} or {PreviousDegrees}.

Types of Attributes (2)

- In general, composite and multi-valued attributes may be nested arbitrarily to any number of levels although this is rare. For example, PreviousDegrees of a STUDENT is a composite multi-valued attribute denoted by {PreviousDegrees (College, Year, Degree, Field)}.

Entity Types and Key Attributes

- Entities with the same basic attributes are grouped or typed into an entity type. For example, the EMPLOYEE entity type or the PROJECT entity type.
- An attribute of an entity type for which each entity must have a unique value is called a key attribute of the entity type. For example, SSN of EMPLOYEE.
- A key attribute may be composite. For example, VehicleTagNumber is a key of the CAR entity type with components (Number, State).
- An entity type may have more than one key. For example, the CAR entity type may have two keys:
 - VehicleIdentificationNumber (popularly called VIN) and
 - VehicleTagNumber (Number, State), also known as license_plate number.

ENTITY SET corresponding to the ENTITY TYPE CAR

CAR

Registration(RegistrationNumber, State), VehicleID, Make, Model, Year, (Color)

car₁

((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 1999, (red, black))

car₂

((ABC 123, NEW YORK), WP9872, Nissan 300ZX, 2-door, 2002, (blue))

car₃

((VSY 720, TEXAS), TD729, Buick LeSabre, 4-door, 2003, (white, blue))

•

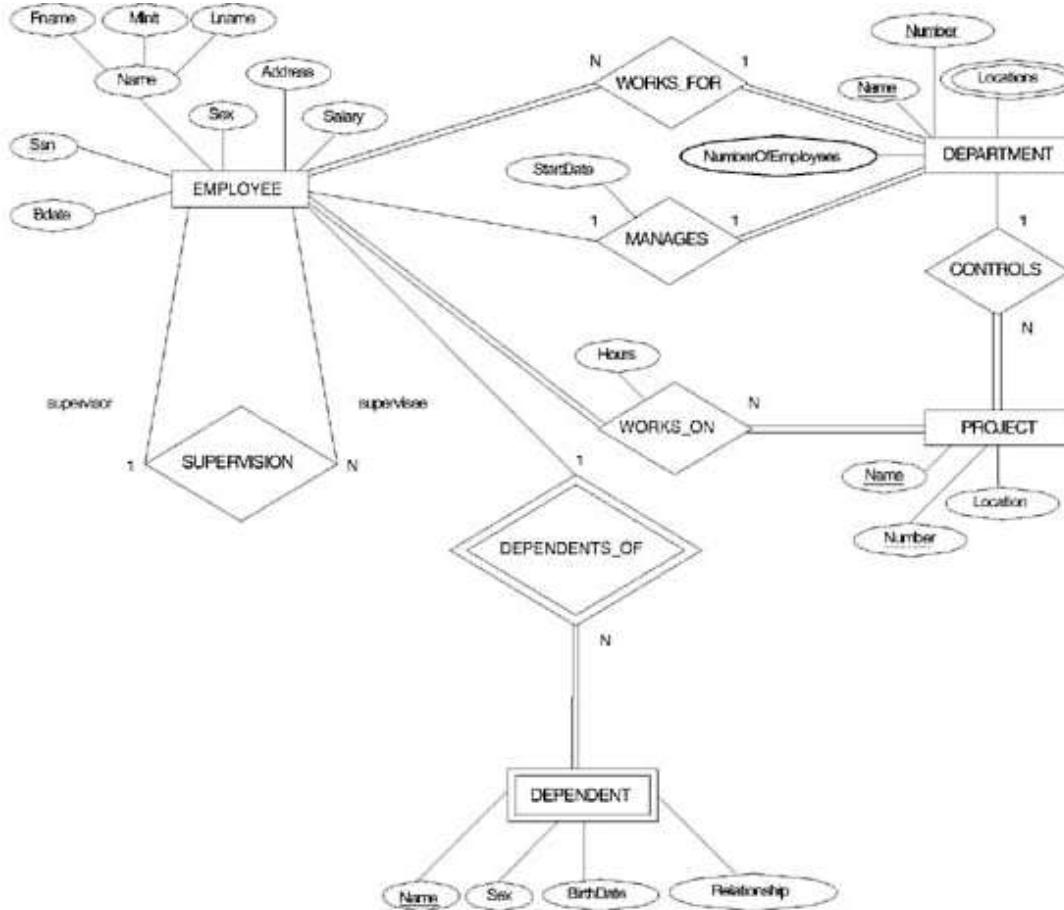
•

•

SUMMARY OF ER-DIAGRAM NOTATION FOR ER SCHEMAS

<u>Symbol</u>	<u>Meaning</u>
	ENTITY TYPE
	WEAK ENTITY TYPE
	RELATIONSHIP TYPE
	IDENTIFYING RELATIONSHIP TYPE
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE
	DERIVED ATTRIBUTE
	TOTAL PARTICIPATION OF E ₂ IN R
	CARDINALITY RATIO 1:N FOR E ₁ :E ₂ IN R
	STRUCTURAL CONSTRAINT (min, max) ON PARTICIPATION OF E IN R

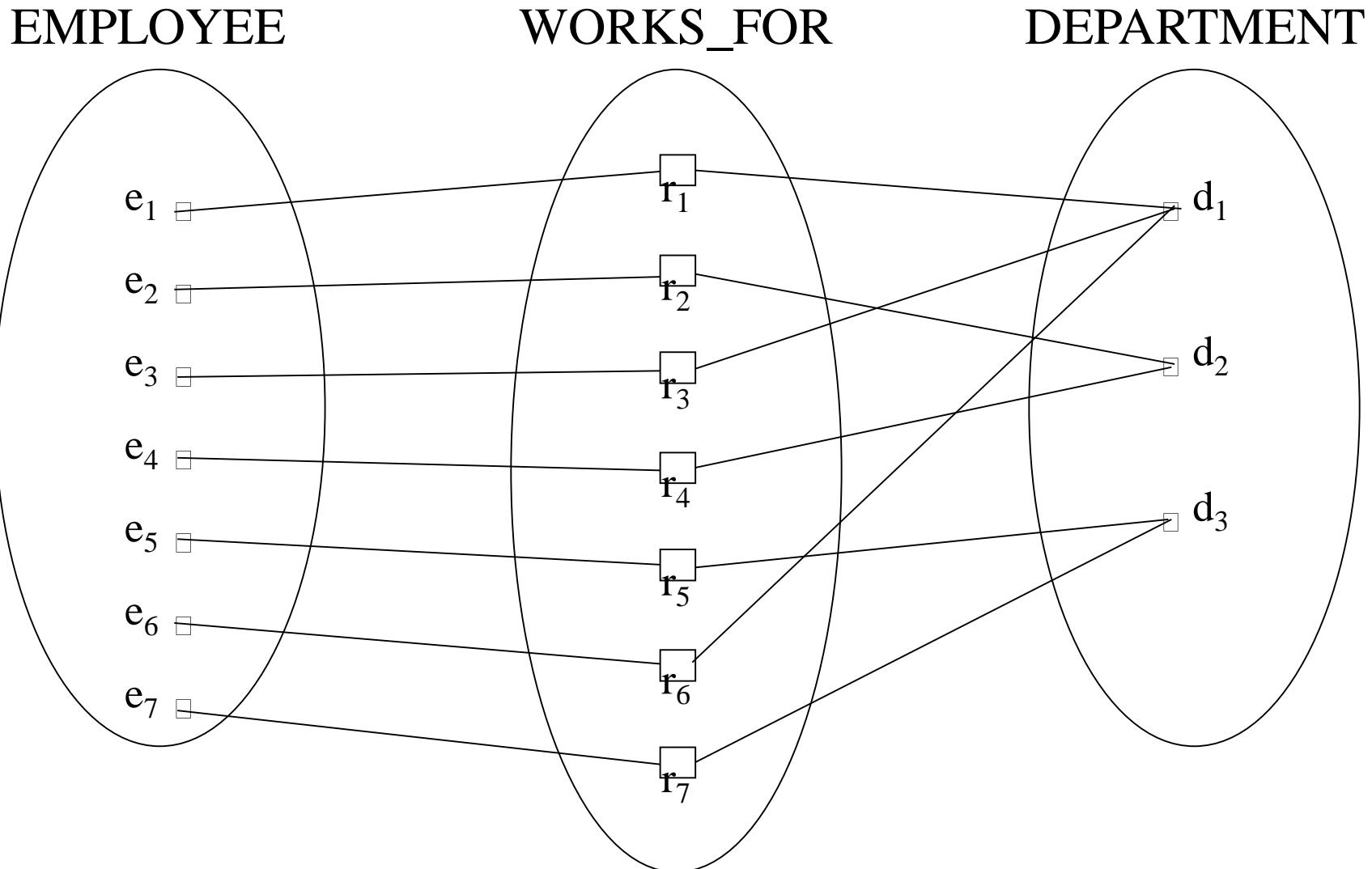
ER DIAGRAM – Entity Types are: EMPLOYEE, DEPARTMENT, PROJECT, DEPENDENT



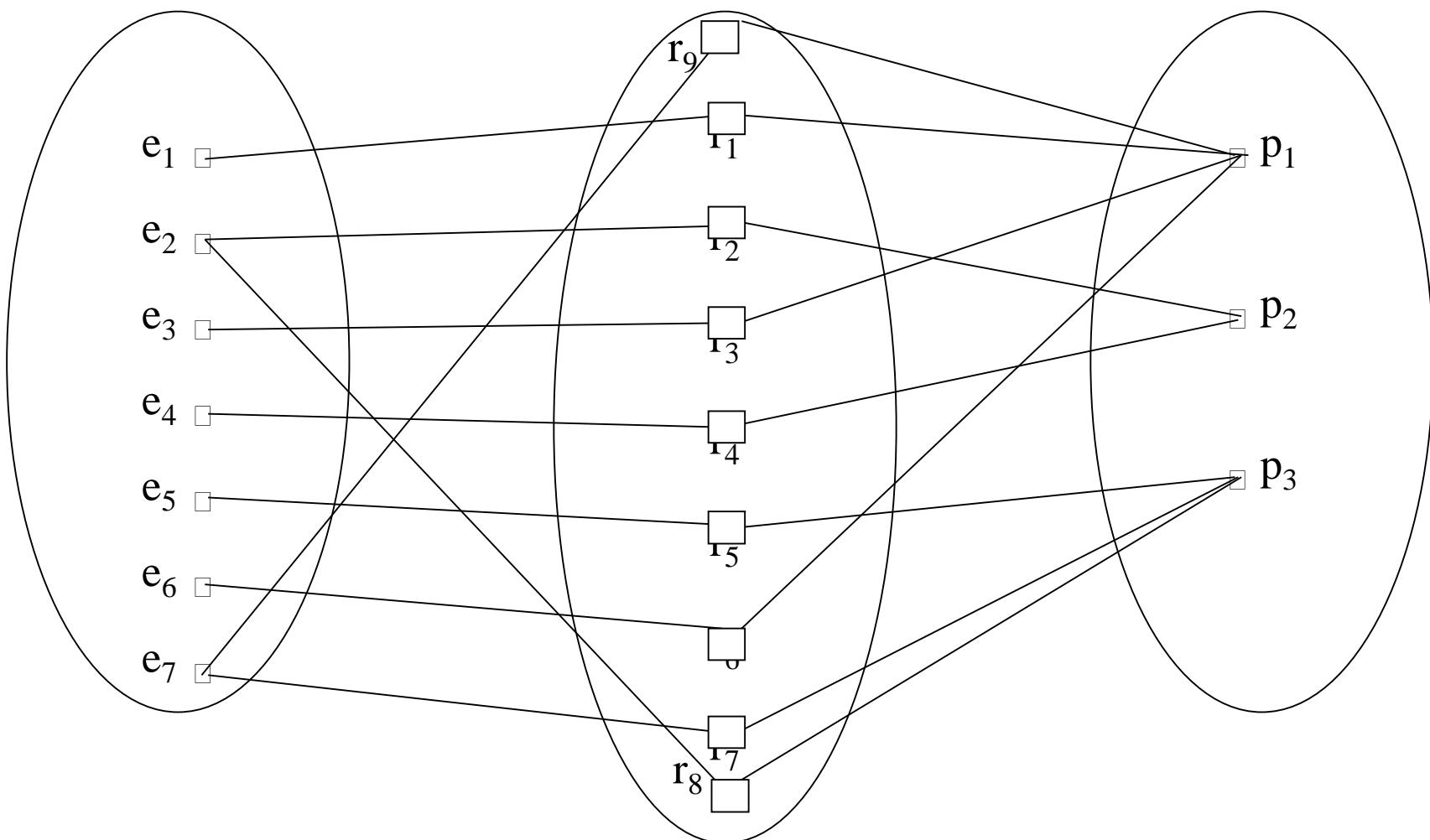
Relationships and Relationship Types (1)

- A relationship relates two or more distinct entities with a specific meaning. For example, EMPLOYEE John Smith works on the ProductX PROJECT or EMPLOYEE Franklin Wong manages the Research DEPARTMENT.
- Relationships of the same type are grouped or typed into a relationship type. For example, the WORKS_ON relationship type in which EMPLOYEES and PROJECTs participate, or the MANAGES relationship type in which EMPLOYEES and DEPARTMENTS participate.
- The degree of a relationship type is the number of participating entity types. Both MANAGES and WORKS_ON are binary relationships.

Example relationship instances of the WORKS_FOR relationship between EMPLOYEE and DEPARTMENT



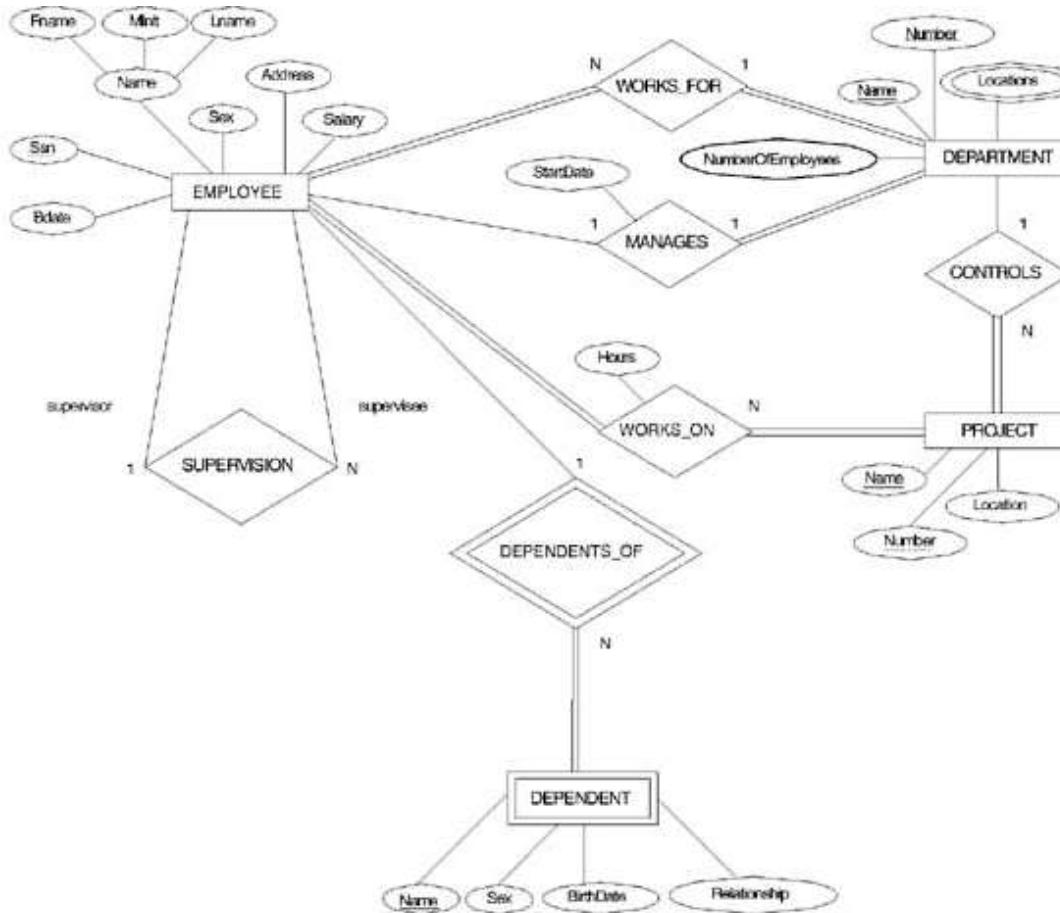
Example relationship instances of the WORKS_ON relationship between EMPLOYEE and PROJECT



Relationships and Relationship Types (2)

- More than one relationship type can exist with the same participating entity types. For example, MANAGES and WORKS_FOR are distinct relationships between EMPLOYEE and DEPARTMENT, but with different meanings and different relationship instances.

ER DIAGRAM – Relationship Types are: WORKS_FOR, MANAGES, WORKS_ON, CONTROLS, SUPERVISION, DEPENDENTS_OF



Weak Entity Types

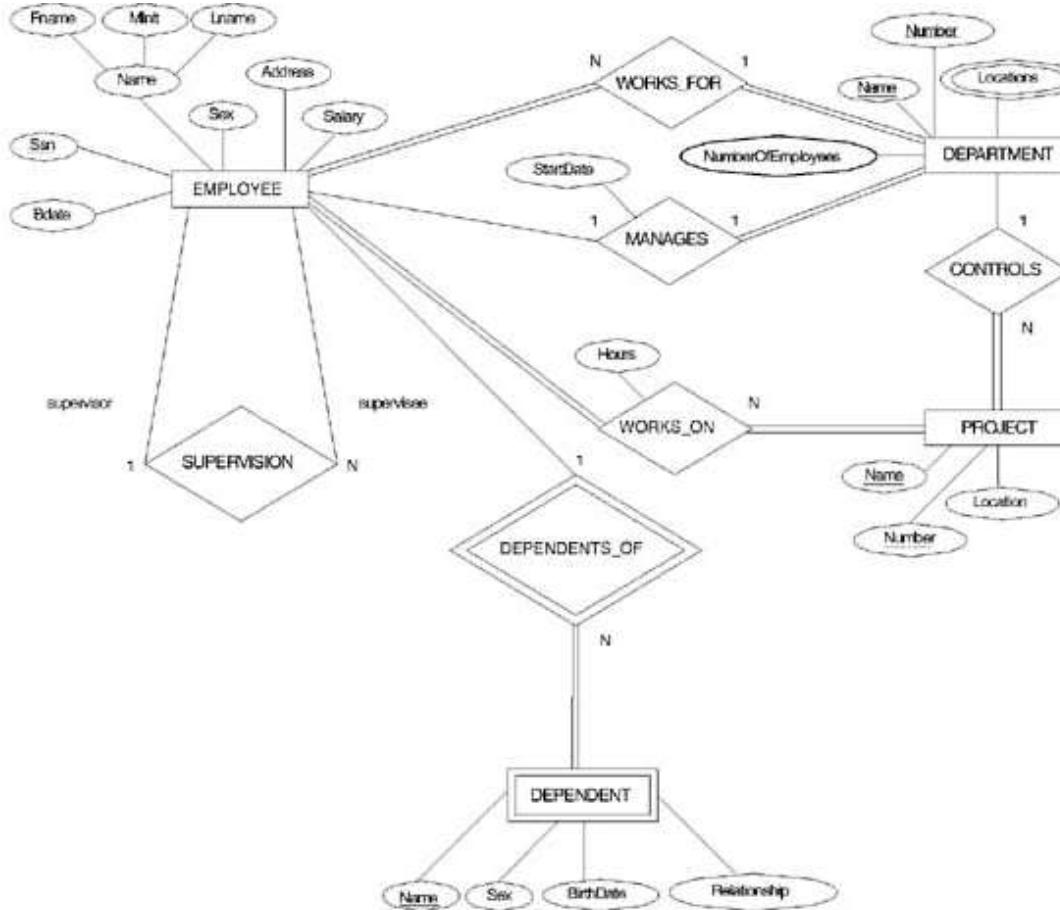
- An entity that does not have a key attribute
- A weak entity must participate in an identifying relationship type with an owner or identifying entity type
- Entities are identified by the combination of:
 - A partial key of the weak entity type
 - The particular entity they are related to in the identifying entity type

Example:

Suppose that a DEPENDENT entity is identified by the dependent's first name and birthdate, *and* the specific EMPLOYEE that the dependent is related to. DEPENDENT is a weak entity type with EMPLOYEE as its identifying entity type via the identifying relationship type
DEPENDENT_OF

Weak Entity Type is: DEPENDENT

Identifying Relationship is: DEPENDENTS_OF

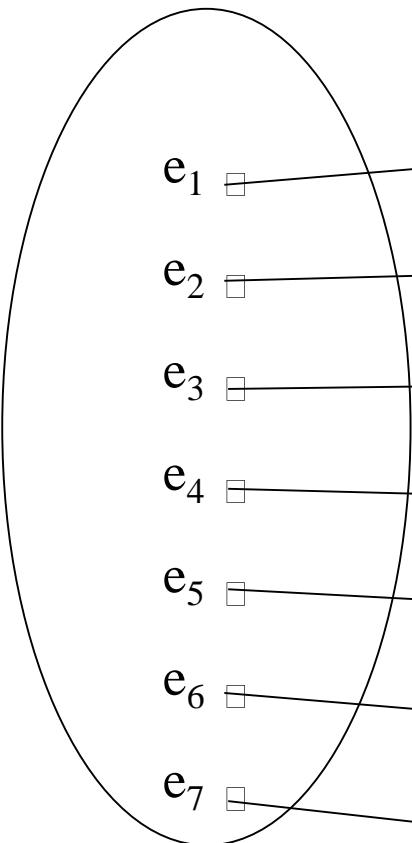


Constraints on Relationships

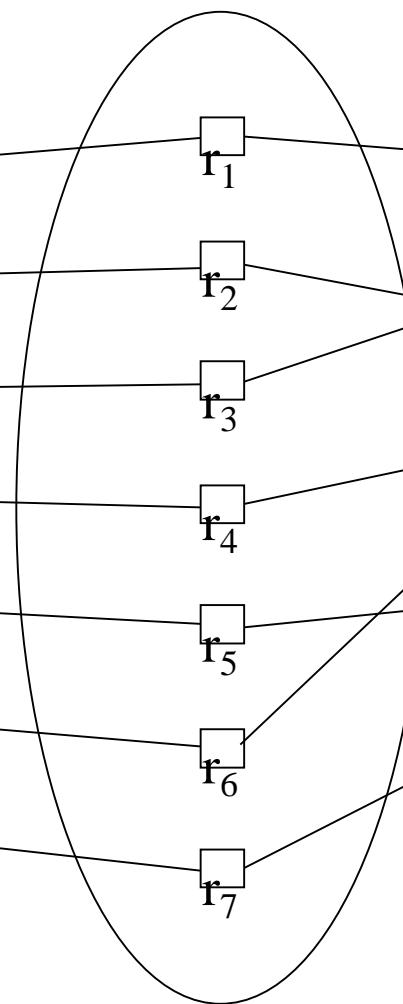
- Constraints on Relationship Types
 - (Also known as ratio constraints)
 - Maximum Cardinality
 - One-to-one (1:1)
 - One-to-many (1:N) or Many-to-one (N:1)
 - Many-to-many
 - Minimum Cardinality (also called participation constraint or existence dependency constraints)
 - zero (optional participation, not existence-dependent)
 - one or more (mandatory, existence-dependent)

Many-to-one (N:1) RELATIONSHIP

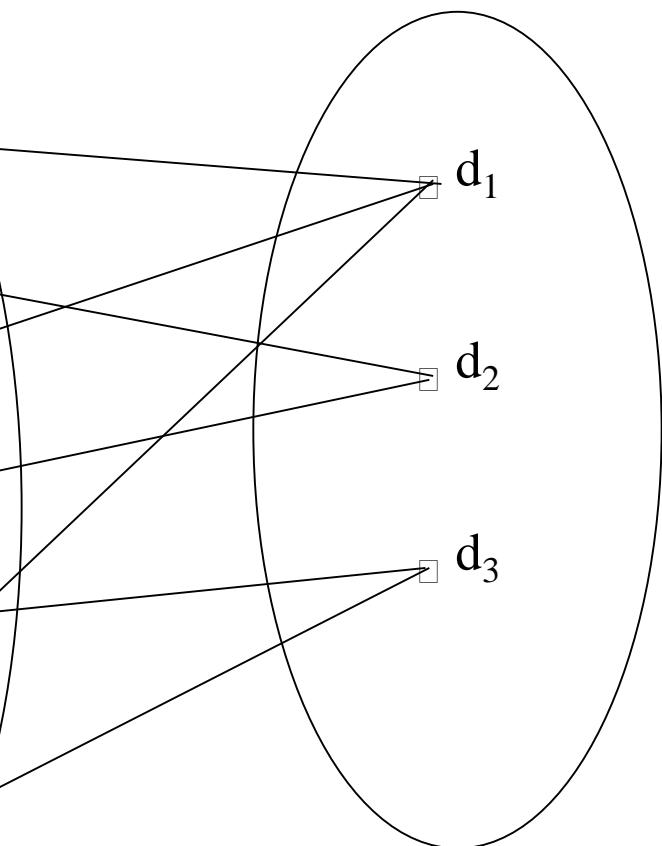
EMPLOYEE



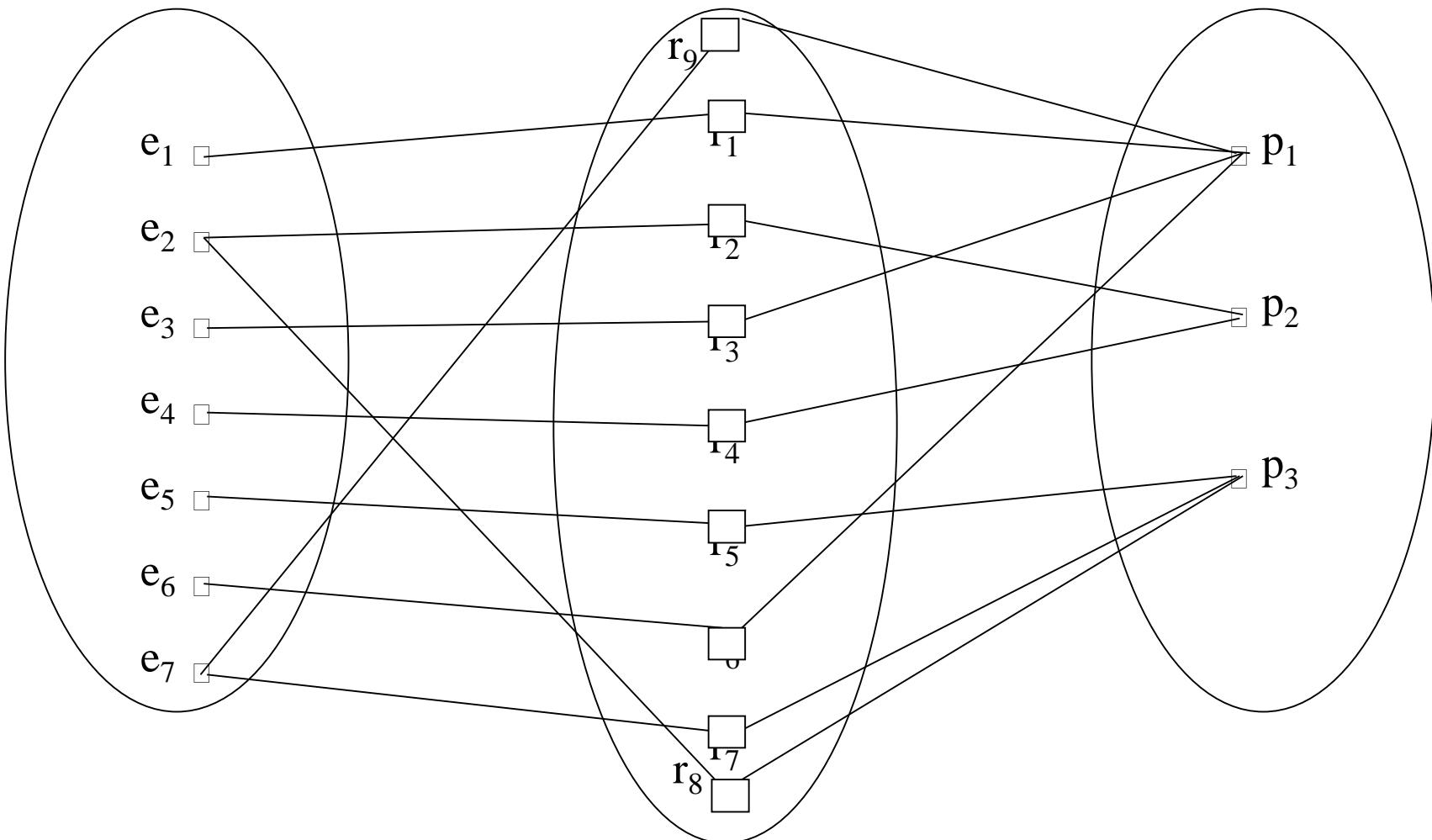
WORKS_FOR



DEPARTMENT



Many-to-many (M:N) RELATIONSHIP



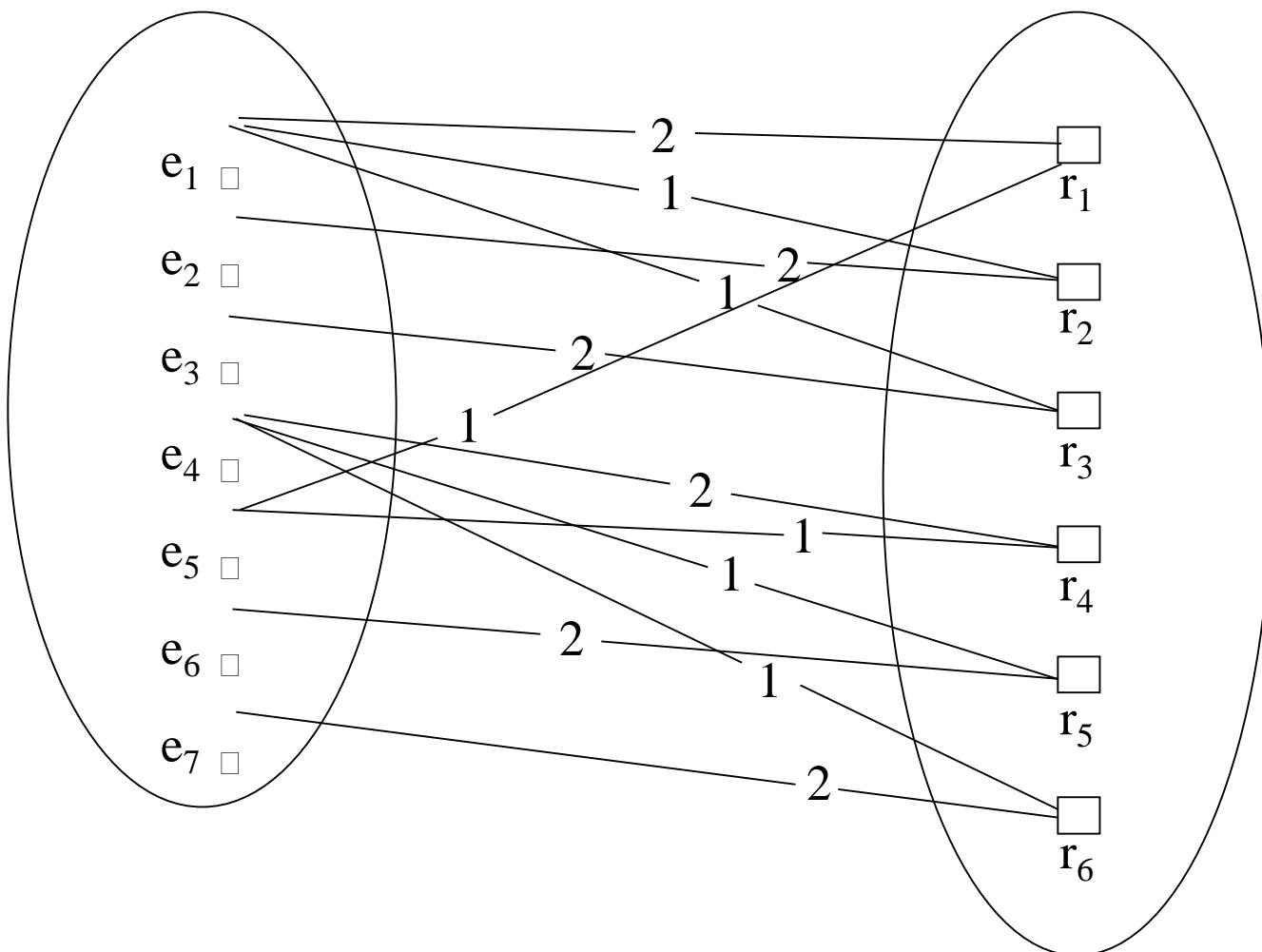
Relationships and Relationship Types (3)

- We can also have a **recursive** relationship type.
- Both participations are same entity type in different roles.
- For example, SUPERVISION relationships between EMPLOYEE (in role of supervisor or boss) and (another) EMPLOYEE (in role of subordinate or worker).
- In following figure, first role participation labeled with 1 and second role participation labeled with 2.
- In ER diagram, need to display role names to distinguish participations.

A RECURSIVE RELATIONSHIP SUPERVISION

EMPLOYEE

SUPERVISION



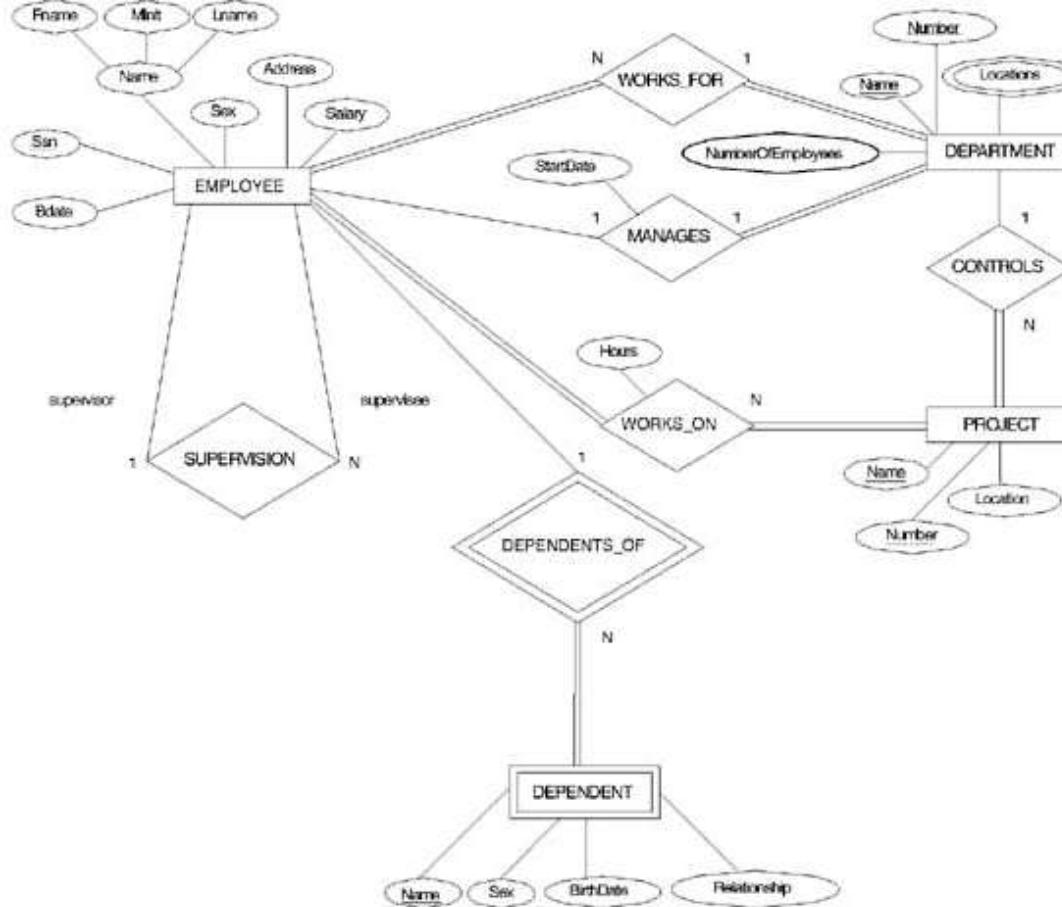
© The Benjamin/Cummings Publishing Company, Inc. 1994, Elmasri/Navathe, Fundamentals of Database Systems, Second Edition

Elmasri and Navathe, *Fundamentals of Database Systems, Fourth Edition*

Copyright © 2004 Pearson Education, Inc.

Chapter 3-81

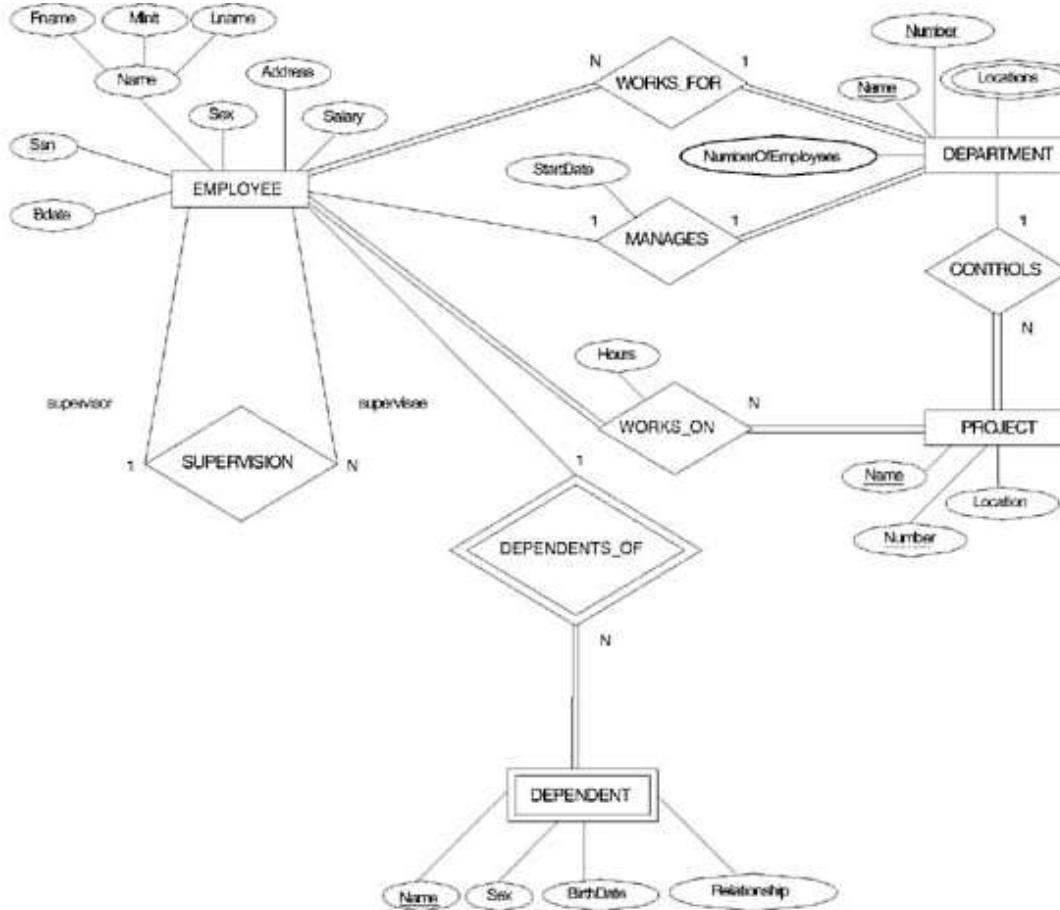
Recursive Relationship Type is: SUPERVISION (participation role names are shown)



Attributes of Relationship types

- A relationship type can have attributes; for example, HoursPerWeek of WORKS_ON; its value for each relationship instance describes the number of hours per week that an EMPLOYEE works on a PROJECT.

Attribute of a Relationship Type is: Hours of WORKS_ON



Structural Constraints – one way to express semantics of relationships

Structural constraints on relationships:

- | **Cardinality ratio** (of a binary relationship): 1:1, 1:N, N:1, or M:N

**SHOWN BY PLACING APPROPRIATE NUMBER ON
THE LINK.**

- | **Participation constraint** (on each participating entity type): total (called *existence dependency*) or partial.

SHOWN BY DOUBLE LINING THE LINK

NOTE: These are easy to specify for Binary Relationship Types.

Alternative (min, max) notation for relationship structural constraints:

- | Specified on *each participation* of an entity type E in a relationship type R
- | Specifies that each entity e in E participates in *at least* min and *at most* max relationship instances in R
- | Default(no constraint): min=0, max=n
- | Must have $\text{min} \leq \text{max}$, $\text{min} \geq 0$, $\text{max} \geq 1$
- | Derived from the knowledge of mini-world constraints

Examples:

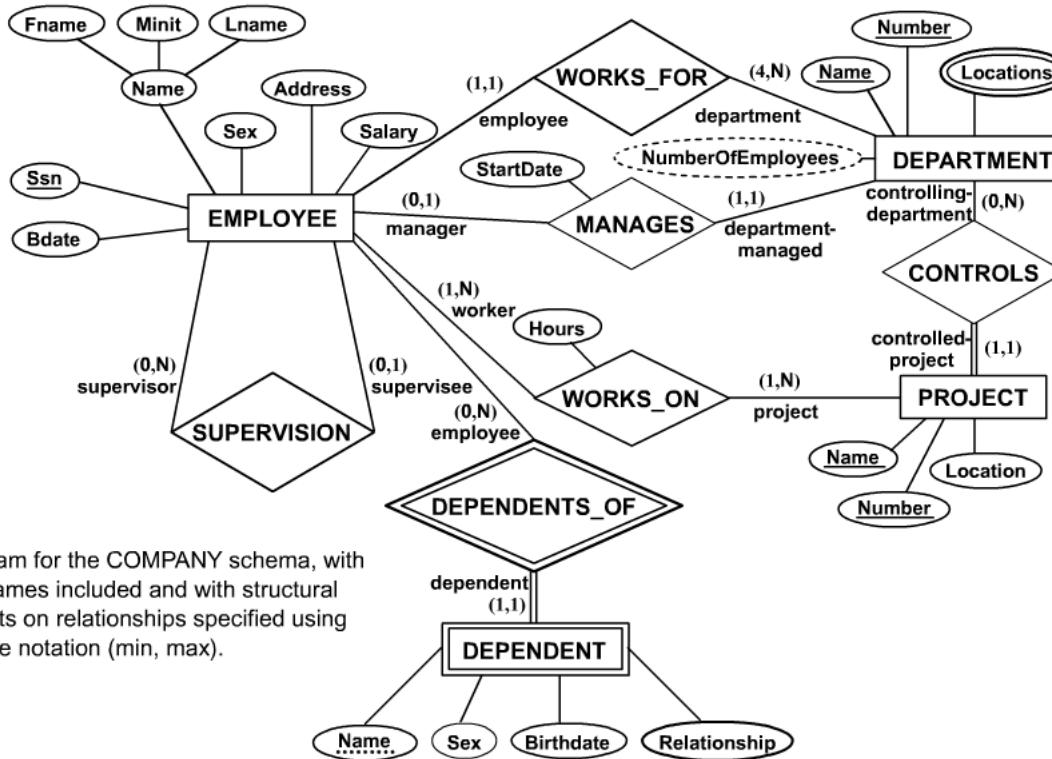
- | A department has *exactly one* manager and an employee can manage *at most one* department.
 - Specify (0,1) for participation of EMPLOYEE in MANAGES
 - Specify (1,1) for participation of DEPARTMENT in MANAGES
- | An employee can work for *exactly one* department but a department can have *any number of employees*.
 - Specify (1,1) for participation of EMPLOYEE in WORKS_FOR
 - Specify (0,n) for participation of DEPARTMENT in WORKS_FOR

The (min,max) notation relationship constraints



COMPANY ER Schema Diagram using (min, max) notation

Alternative ER Notations



ER diagram for the COMPANY schema, with all role names included and with structural constraints on relationships specified using alternative notation (min, max).

Relationships of Higher Degree

- | Relationship types of degree 2 are called **binary**
- | Relationship types of degree 3 are called **ternary** and of degree n are called **n-ary**
- | In general, an n-ary relationship *is not* equivalent to n binary relationships
- | Higher-order relationships discussed further in Chapter 4

Data Modeling Tools

A number of popular tools that cover conceptual modeling and mapping into relational schema design. Examples: ERWin, S- Designer (Enterprise Application Suite), ER- Studio, etc.

POSITIVES: serves as documentation of application requirements, easy user interface - mostly graphics editor support

Problems with Current Modeling Tools

● DIAGRAMMING

- Poor conceptual meaningful notation.
- To avoid the problem of layout algorithms and aesthetics of diagrams, they prefer boxes and lines and do nothing more than represent (primary-foreign key) relationships among resulting tables.(a few exceptions)

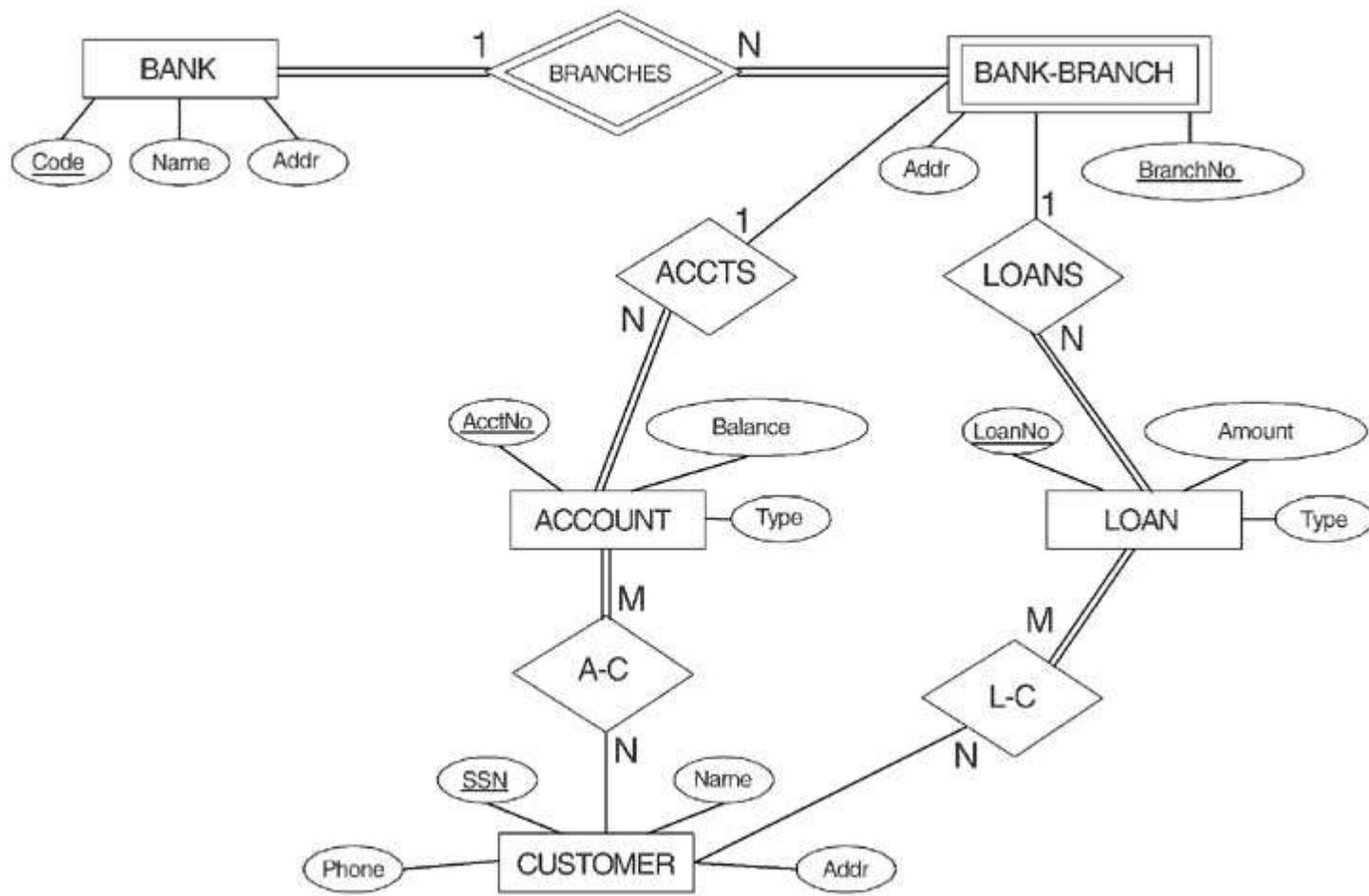
● METHODOLOGY

- lack of built-in methodology support.
- poor tradeoff analysis or user-driven design preferences.
- poor design verification and suggestions for improvement.

Some of the Currently Available Automated Database Design Tools

COMPANY	TOOL	FUNCTIONALITY
Embarcadero Technologies	ER Studio	Database Modeling in ER and IDEF1X
	DB Artisan	Database administration and space and security management
Oracle	Developer 2000 and Designer 2000	Database modeling, application development
Popkin Software	System Architect 2001	Data modeling, object modeling, process modeling, structured analysis/design
Platinum Technology	Platinum Enterprise Modeling Suite: Erwin, BPWin, Paradigm Plus	Data, process, and business component modeling
Persistence Inc.	Pwertier	Mapping from O-O to relational model
Rational	Rational Rose	Modeling in UML and application generation in C++ and JAVA
Rogue Ware	RW Metro	Mapping from O-O to relational model
Resolution Ltd.	Xcase	Conceptual modeling up to code maintenance
Sybase	Enterprise Application Suite	Data modeling, business logic modeling
Visio	Visio Enterprise	Data modeling, design and reengineering Visual Basic and Visual C++

ER DIAGRAM FOR A BANK DATABASE



© The Benjamin/Cummings Publishing Company, Inc. 1994, Elmasri/Navathe, Fundamentals of Database Systems, Second Edition

PROBLEM with ER notation

THE ENTITY RELATIONSHIP MODEL IN
ITS ORIGINAL FORM DID NOT
SUPPORT THE SPECIALIZATION/
GENERALIZATION ABSTRACTIONS

Extended Entity-Relationship (EER) Model

- Incorporates Set-subset relationships
- Incorporates Specialization/Generalization Hierarchies

NEXT CHAPTER ILLUSTRATES HOW THE ER MODEL CAN BE EXTENDED WITH

- Set-subset relationships and Specialization/Generalization Hierarchies and how to display them in EER diagrams

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 5

The Relational Data Model and Relational Database Constraints



Chapter Outline

- Relational Model Concepts
- Relational Model Constraints and Relational Database Schemas
- Update Operations and Dealing with Constraint Violations

Relational Model Concepts

- The relational Model of Data is based on the concept of a Relation.
- A Relation is a mathematical concept based on the ideas of sets.
- The strength of the relational approach to data management comes from the formal foundation provided by the theory of relations.
- We review the essentials of the relational approach in this chapter.

Relational Model Concepts

- The model was first proposed by Dr. E.F. Codd of IBM in 1970 in the following paper:
"A Relational Model for Large Shared Data Banks," Communications of the ACM, June 1970.

The above paper caused a major revolution in the field of Database management and earned Ted Codd the coveted ACM Turing Award.

INFORMAL DEFINITIONS

- **RELATION:** A table of values

- A relation may be thought of as a **set of rows**.
- A relation may alternately be thought of as a **set of columns**.
- Each row represents a fact that corresponds to a real-world **entity** or **relationship**.
- Each row has a value of an item or set of items that uniquely identifies that row in the table.
- Sometimes row-ids or sequential numbers are assigned to identify the rows in the table.
- Each column typically is called by its column name or column header or attribute name.

FORMAL DEFINITIONS

- A **Relation** may be defined in multiple ways.
- The **Schema** of a Relation: R (A_1, A_2, \dots, A_n)

Relation schema R is defined over **attributes** A_1, A_2, \dots, A_n

For Example -

CUSTOMER (Cust-id, Cust-name, Address, Phone#)

Here, CUSTOMER is a relation defined over the four attributes Cust-id, Cust-name, Address, Phone#, each of which has a **domain** or a set of valid values. For example, the domain of Cust-id is 6 digit numbers.

FORMAL DEFINITIONS

- A **tuple** is an ordered set of values
- Each value is derived from an appropriate domain.
- Each row in the CUSTOMER table may be referred to as a tuple in the table and would consist of four values.
- <632895, "John Smith", "101 Main St. Atlanta, GA 30332", "(404) 894-2000"> is a tuple belonging to the CUSTOMER relation.
- A relation may be regarded as a *set of tuples* (rows).
- Columns in a table are also called attributes of the relation.

FORMAL DEFINITIONS

- A **domain** has a logical definition: e.g., “USA_phone_numbers” are the set of 10 digit phone numbers valid in the U.S.
- A domain may have a data-type or a format defined for it. The USA_phone_numbers may have a format: (ddd)-ddd-dddd where each d is a decimal digit. E.g., Dates have various formats such as monthname, date, year or yyyy-mm-dd, or dd mm,yyyy etc.
- An attribute designates the **role** played by the domain. E.g., the domain Date may be used to define attributes “Invoice-date” and “Payment-date”.

FORMAL DEFINITIONS

- The relation is formed over the cartesian product of the sets; each set has values from a domain; that domain is used in a specific role which is conveyed by the attribute name.
- For example, attribute Cust-name is defined over the domain of strings of 25 characters. The role these strings play in the CUSTOMER relation is that of the name of customers.
- Formally,

Given $R(A_1, A_2, \dots, A_n)$

$$r(R) \subset \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$$

- R: schema of the relation
- r of R: a specific "value" or population of R.
- R is also called the **intension** of a relation
- r is also called the **extension** of a relation

FORMAL DEFINITIONS

- Let $S_1 = \{0,1\}$
- Let $S_2 = \{a,b,c\}$
- Let $R \subset S_1 \times S_2$
- Then for example: $r(R) = \{\langle 0, a \rangle, \langle 0, b \rangle, \langle 1, c \rangle\}$ is one possible “state” or “population” or “extension” r of the relation R , defined over domains S_1 and S_2 . It has three tuples.

DEFINITION SUMMARY

Informal Terms

Table

Column

Row

Values in a column

Table Definition

Populated Table

Formal Terms

Relation

Attribute/Domain

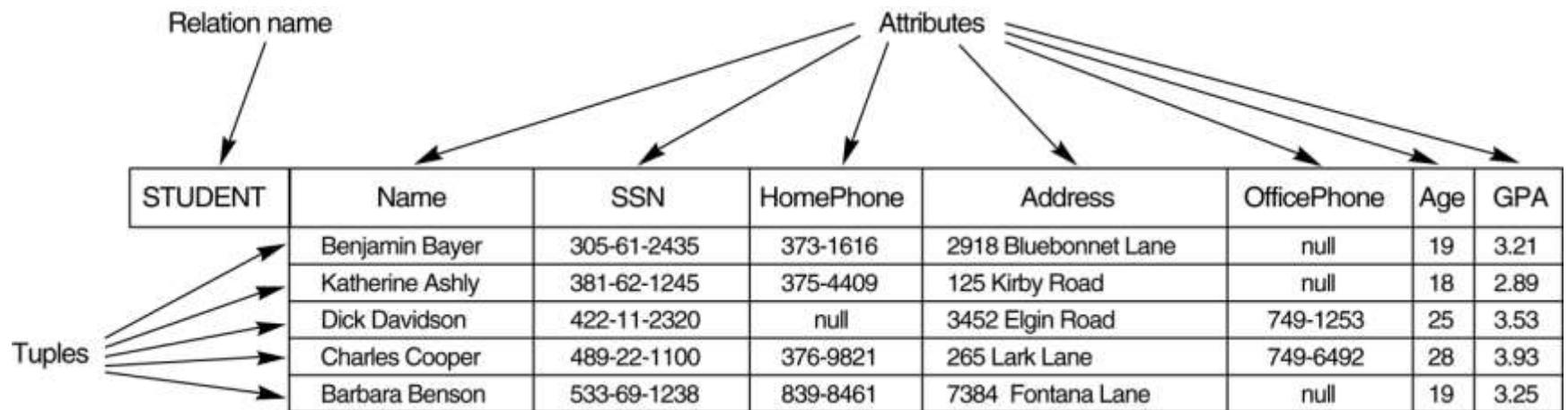
Tuple

Domain

Schema of a Relation

Extension

Example - Figure 5.1



CHARACTERISTICS OF RELATIONS

- **Ordering of tuples in a relation $r(R)$:** The tuples are *not* considered to be ordered, even though they appear to be in the tabular form.
- **Ordering of attributes in a relation schema R (and of values within each tuple):** We will consider the attributes in $R(A_1, A_2, \dots, A_n)$ and the values in $t = <v_1, v_2, \dots, v_n>$ to be *ordered*.
(However, a more general *alternative definition* of relation does not require this ordering).
- **Values in a tuple:** All values are considered *atomic* (indivisible). A special **null** value is used to represent values that are unknown or inapplicable to certain tuples.

CHARACTERISTICS OF RELATIONS

● Notation:

- We refer to **component values** of a tuple t by $t[A_i] = v_i$ (the value of attribute A_i for tuple t).

Similarly, $t[A_u, A_v, \dots, A_w]$ refers to the subtuple of t containing the values of attributes A_u, A_v, \dots, A_w , respectively.

CHARACTERISTICS OF RELATIONS- Figure 5.2

STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21

Relational Integrity Constraints

- Constraints are *conditions* that must hold on *all* valid relation instances. There are three main types of constraints:
 1. Key constraints
 2. Entity integrity constraints
 3. Referential integrity constraints

Key Constraints

- **Superkey of R:** A set of attributes SK of R such that no two tuples *in any valid relation instance r(R)* will have the same value for SK. That is, for any distinct tuples t1 and t2 in r(R), $t1[SK] \neq t2[SK]$.
- **Key of R:** A "minimal" superkey; that is, a superkey K such that removal of any attribute from K results in a set of attributes that is not a superkey.

Example: The CAR relation schema:

CAR(State, Reg#, SerialNo, Make, Model, Year)

has two keys Key1 = {State, Reg#}, Key2 = {SerialNo}, which are also superkeys. {SerialNo, Make} is a superkey but *not* a key.

- If a relation has *several candidate keys*, one is chosen arbitrarily to be the **primary key**. The primary key attributes are *underlined*.

Key Constraints

Figure 5.4 The CAR relation with two candidate keys:
LicenseNumber and EngineSerialNumber.

CAR	<u>LicenseNumber</u>	EngineSerialNumber	Make	Model	Year
	Texas ABC-739	A69352	Ford	Mustang	96
	Florida TVP-347	B43696	Oldsmobile	Cutlass	99
	New York MPO-22	X83554	Oldsmobile	Delta	95
	California 432-TFY	C43742	Mercedes	190-D	93
	California RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

Entity Integrity

- **Relational Database Schema:** A set S of relation schemas that belong to the same database. S is the *name* of the **database**.

$$S = \{R_1, R_2, \dots, R_n\}$$

- **Entity Integrity:** The *primary key attributes* PK of each relation schema R in S cannot have null values in any tuple of $r(R)$. This is because primary key values are used to *identify* the individual tuples.

$t[PK] \neq \text{null}$ for any tuple t in $r(R)$

- Note: Other attributes of R may be similarly constrained to disallow null values, even though they are not members of the primary key.

Referential Integrity

- A constraint involving *two* relations (the previous constraints involve a *single* relation).
- Used to specify a *relationship* among tuples in two relations: the **referencing relation** and the **referenced relation**.
- Tuples in the *referencing relation* R_1 have attributes FK (called **foreign key** attributes) that reference the primary key attributes PK of the *referenced relation* R_2 . A tuple t_1 in R_1 is said to **reference** a tuple t_2 in R_2 if $t_1[FK] = t_2[PK]$.
- A referential integrity constraint can be displayed in a relational database schema as a directed arc from $R_1.FK$ to R_2 .

Referential Integrity Constraint

Statement of the constraint

The value in the foreign key column (or columns) FK of the the **referencing relation** R_1 can be either:

- (1) a value of an existing primary key value of the corresponding primary key PK in the **referenced relation** R_2 , or..
- (2) a null.

In case (2), the FK in R_1 should not be a part of its own primary key.

Other Types of Constraints

Semantic Integrity Constraints:

- based on application semantics and cannot be expressed by the model per se
- E.g., “the max. no. of hours per employee for all projects he or she works on is 56 hrs per week”
- A *constraint specification language* may have to be used to express these
- SQL-99 allows triggers and ASSERTIONS to allow for some of these

Figure 5.5 Schema diagram for the COMPANY relational database schema; the primary keys are underlined.

EMPLOYEE									
FNAME	MINIT	LNAME	<u>SSN</u>	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
DEPARTMENT									
DNAME	<u>DNUMBER</u>		MGRSSN	MGRSTARTDATE					
DEPT_LOCATIONS									
DNUMBER		DLOCATION							
PROJECT									
PNAME	<u>PNUMBER</u>	PLOCATION		DNUM					
WORKS_ON									
ESSN	PNO	HOURS							
DEPENDENT									
ESSN	DEPENDENT_NAME		SEX	BDATE	RELATIONSHIP				

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Figure 5.6 One possible relational database state corresponding to the COMPANY schema.

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSEN	DNO
John	Smith	J	Smith	123456789	1965-01-09	231 Finchln, Houston, TX	M	30000	333455555	5
Franklin	Wong		Wong	333456789	1965-12-08	638 Vow, Houston, TX	M	40000	867654321	5
Alicia	Zelena		Zelena	666667777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	Walker	J	Walker	667754321	1971-06-20	251 Hwy, Dallas, TX	F	42000	867654321	4
Ramesh	Narayan		Narayan	668888444	1968-09-15	975 Elm Oak, Humble, TX	M	36000	333455555	5
Joyce	English		English	433453453	1972-07-31	931 Rice, Houston, TX	F	28000	333455555	5
Ahmad	Jabber		Jabber	997987987	1969-03-29	880 Dallas, Houston, TX	M	27000	987654321	4
James	Borg		Borg	660006666	1970-11-10	430 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
		Houston
		Stafford
		Biloxi
		Sugartown

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research	R&D	3	333455555	1988-05-22
Administration	Admin	4	987654321	1995-01-01
Headquarters	HQ	1	666666666	1981-06-19

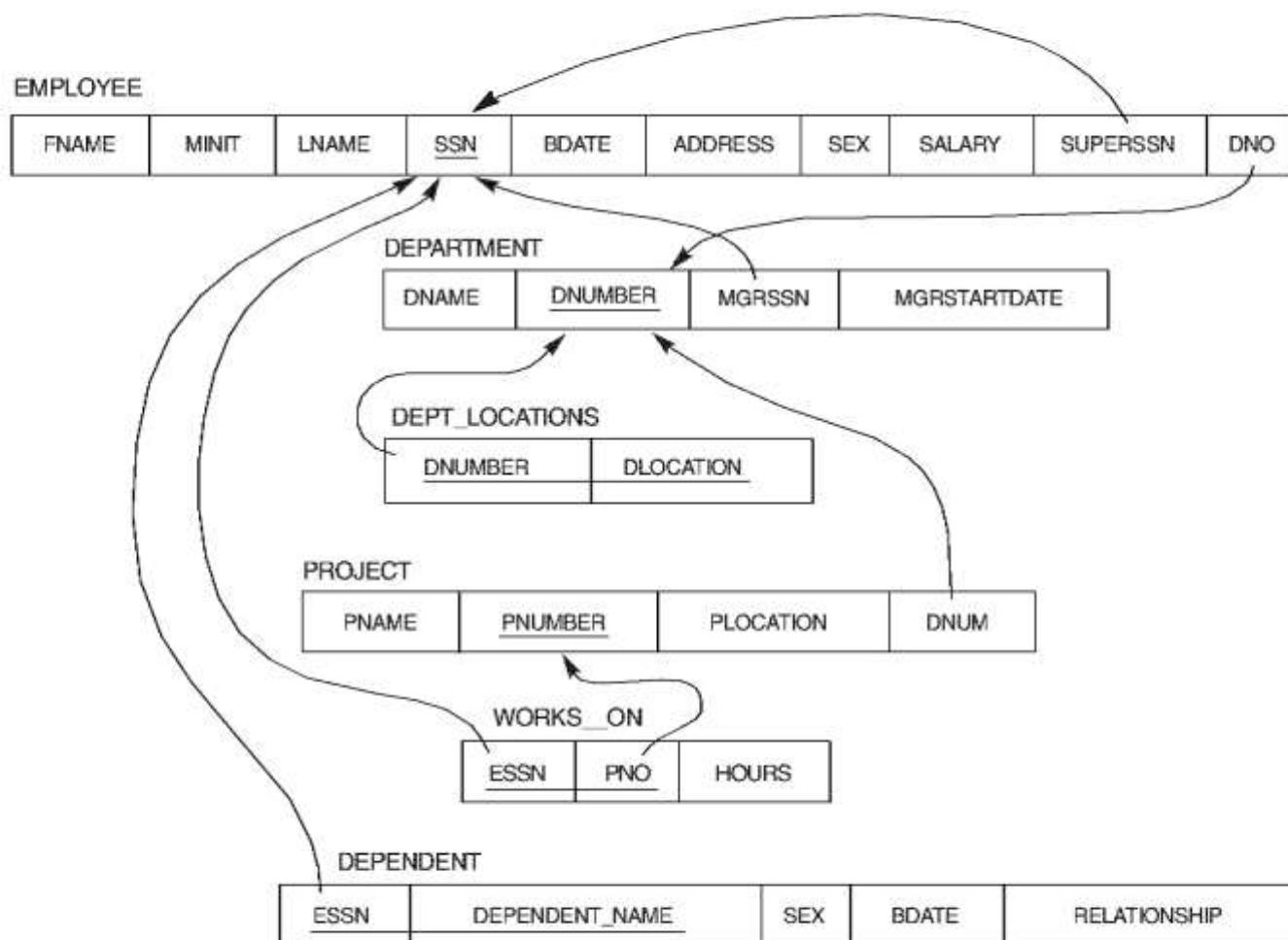
WORKS ON	ESSN	PNO	HOURS
123456789	1	32.5	
123456789	2	7.5	
666664444	3	40.0	
433453453	1	20.0	
433453453	2	20.0	
333455555	2	10.0	
333455555	3	10.0	
333455555	10	10.0	
333455555	20	10.0	
999997777	30	30.0	
999997777	10	10.0	
987987987	10	35.0	
987987987	30	5.0	
987654321	20	20.0	
987654321	20	15.0	
666666666	20	null	

PROJECT	PNAME	PNUMBER	PLCATION	DNUM
ProductX	X	1	Biloxi	5
ProductY	Y	2	Sugartown	5
ProductZ	Z	3	Houston	5
Computerization	C	10	Stafford	4
Reorganization	R	20	Houston	1
New benefits	N	30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
333455555	Alicia		F	1986-04-05	DAUGHTER
333455555	Theodore		M	1983-10-25	SON
333455555	Jay		F	1989-05-01	SPOUSE
987654321	Abner		M	1942-02-28	SPOUSE
123456789	Michael		M	1988-01-04	SON
123456789	Alan		F	1986-12-30	DAUGHTER
123456789	Elizabeth		F	1987-05-05	SPOUSE

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Figure 5.7 Referential integrity constraints displayed on the COMPANY relational database schema diagram.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Update Operations on Relations

- INSERT a tuple.
- DELETE a tuple.
- MODIFY a tuple.

- Integrity constraints should not be violated by the update operations.
- Several update operations may have to be grouped together.
- Updates may *propagate* to cause other updates automatically. This may be necessary to maintain integrity constraints.

Update Operations on Relations

- In case of integrity violation, several actions can be taken:
 - Cancel the operation that causes the violation (REJECT option)
 - Perform the operation but inform the user of the violation
 - Trigger additional updates so the violation is corrected (CASCADE option, SET NULL option)
 - Execute a user-specified error-correction routine

In-Class Exercise

(Taken from Exercise 5.15)

Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(SSN, Name, Major, Bdate)

COURSE(Course#, Cname, Dept)

ENROLL(SSN, Course#, Quarter, Grade)

BOOK_ADOPTION(Course#, Quarter, Book_ISBN)

TEXT(Book_ISBN, Book_Title, Publisher, Author)

Draw a relational schema diagram specifying the foreign keys for this schema.

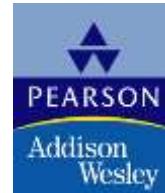
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 6

The Relational Algebra and Calculus



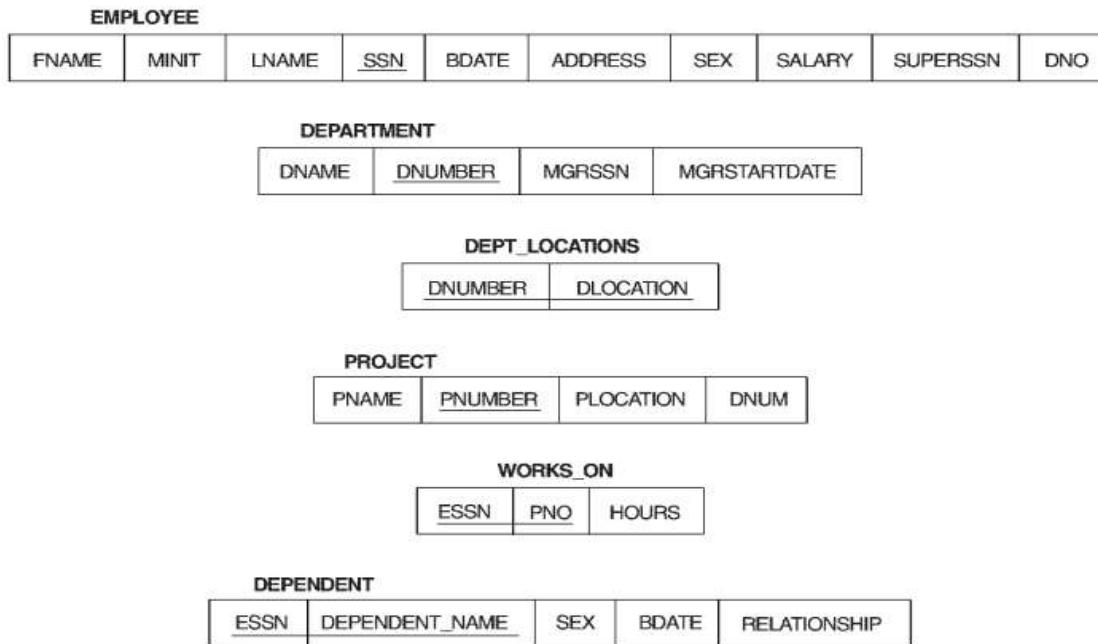
Chapter Outline

- Example Database Application (COMPANY)
- Relational Algebra
 - Unary Relational Operations
 - Relational Algebra Operations From Set Theory
 - Binary Relational Operations
 - Additional Relational Operations
 - Examples of Queries in Relational Algebra
- Relational Calculus
 - Tuple Relational Calculus
 - Domain Relational Calculus
- Overview of the QBE language (appendix D)

Database State for COMPANY

All examples discussed below refer to the COMPANY database shown here.

Figure 7.5 Schema diagram for the COMPANY relational database schema; the primary keys are underlined.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Relational Algebra

- The basic set of operations for the relational model is known as the relational algebra. These operations enable a user to specify basic retrieval requests.
- The result of a retrieval is a new relation, which may have been formed from one or more relations. The **algebra operations** thus produce new relations, which can be further manipulated using operations of the same algebra.
- A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

Unary Relational Operations

● SELECT Operation

SELECT operation is used to select a *subset* of the tuples from a relation that satisfy a **selection condition**. It is a filter that keeps only those tuples that satisfy a qualifying condition – those satisfying the condition are selected while others are discarded.

Example: To select the EMPLOYEE tuples whose department number is four or those whose salary is greater than \$30,000 the following notation is used:

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

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

In general, the select operation is denoted by $\sigma_{<\text{selection condition}>} (R)$ where the symbol σ (sigma) is used to denote the select operator, and the selection condition is a Boolean expression specified on the attributes of relation R

Unary Relational Operations

SELECT Operation Properties

- The SELECT operation $\sigma_{<\text{selection condition}>}(R)$ produces a relation S that has the same schema as R
- The SELECT operation σ is **commutative**; i.e.,
$$\sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>}(R)) = \sigma_{<\text{condition2}>}(\sigma_{<\text{condition1}>}(R))$$
- A cascaded SELECT operation **may be applied in any order**; i.e.,
$$\begin{aligned} & \sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>}(\sigma_{<\text{condition3}>}(R))) \\ &= \sigma_{<\text{condition2}>}(\sigma_{<\text{condition3}>}(\sigma_{<\text{condition1}>}(R))) \end{aligned}$$
- A cascaded SELECT operation may be replaced by a single selection with a conjunction of all the conditions; i.e.,
$$\begin{aligned} & \sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>}(\sigma_{<\text{condition3}>}(R))) \\ &= \sigma_{<\text{condition1}> \text{ AND } <\text{condition2}> \text{ AND } <\text{condition3}>}(R) \end{aligned}$$

Unary Relational Operations (cont.)

Figure 7.8 Results of SELECT and PROJECT operations.

(a) $\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(\text{EMPLOYEE})$.

(b) $\pi_{\text{LNAME}, \text{FNAME}, \text{SALARY}}(\text{EMPLOYEE})$. (c) $\pi_{\text{SEX}, \text{SALARY}}(\text{EMPLOYEE})$

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak, Humble, TX	M	38000	333445555	5

(b)

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

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Unary Relational Operations (cont.)

● PROJECT Operation

This operation selects certain *columns* from the table and discards the other columns. The PROJECT creates a vertical partitioning – one with the needed columns (attributes) containing results of the operation and other containing the discarded Columns.

Example: To list each employee's first and last name and salary, the following is used:

$$\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$$

The general form of the project operation is $\pi_{<\text{attribute list}>}(\text{R})$ where π (pi) is the symbol used to represent the project operation and $<\text{attribute list}>$ is the desired list of attributes from the attributes of relation R.

The project operation *removes any duplicate tuples*, so the result of the project operation is a set of tuples and hence a valid relation.

Unary Relational Operations (cont.)

PROJECT Operation Properties

- The number of tuples in the result of projection $\pi_{<\text{list}>}(\text{R})$ is always less or equal to the number of tuples in R.
- If the list of attributes includes a key of R, then the number of tuples is equal to the number of tuples in R.
- $\pi_{<\text{list1}>}(\pi_{<\text{list2}>}(\text{R})) = \pi_{<\text{list1}>}(\text{R})$ as long as $<\text{list2}>$ contains the attributes in $<\text{list2}>$

Unary Relational Operations (cont.)

Figure 7.8 Results of SELECT and PROJECT operations.

(a) $\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(\text{EMPLOYEE})$.

(b) $\pi_{\text{LNAME}, \text{FNAME}, \text{SALARY}}(\text{EMPLOYEE})$. (c) $\pi_{\text{SEX}, \text{SALARY}}(\text{EMPLOYEE})$

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak, Humble, TX	M	38000	333445555	5

(b)

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

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Unary Relational Operations (cont.)

● Rename Operation

We may want 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.

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 as follows:

$$\pi_{\text{FNAME}, \text{LNAME}, \text{SALARY}}(\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$$

OR We can explicitly show the sequence of operations, giving a name to each intermediate relation:

$$\text{DEP5_EMPS} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{SALARY}} (\text{DEP5_EMPS})$$

Unary Relational Operations (cont.)

- **Rename Operation (cont.)**

The rename operator is ρ

The general Rename operation can be expressed by any of the following forms:

- $\rho_{S(B_1, B_2, \dots, B_n)}(R)$ is a renamed relation S based on R with column names B_1, B_1, \dots, B_n .
- $\rho_S(R)$ is a renamed relation S based on R (which does not specify column names).
- $\rho_{(B_1, B_2, \dots, B_n)}(R)$ is a renamed relation with column names B_1, B_1, \dots, B_n which does not specify a new relation name.

Unary Relational Operations (cont.)

Figure 7.9 Results of relational algebra expressions.

(a) $\pi_{\text{LNAME}, \text{FNAME}, \text{SALARY}} (\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$. (b) The same expression using intermediate relations and renaming of attributes.

(a)

FNAME	LNAME	SALARY
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	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	686884444	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	

	FIRSTNAME	LASTNAME	SALARY
	John	Smith	30000
	Franklin	Wong	40000
	Ramesh	Narayan	38000
	Joyce	English	25000

Relational Algebra Operations From Set Theory

● UNION Operation

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.

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, we can use the union operation as follows:

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

RESULT1 $\leftarrow \pi_{\text{SSN}}(\text{DEP5_EMPS})$

RESULT2(SSN) $\leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5_EMPS})$

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

The union operation produces the tuples that are in either RESULT1 or RESULT2 or both. The two operands must be “type compatible”.

Relational Algebra Operations From Set Theory

● Type Compatibility

- The operand relations $R_1(A_1, A_2, \dots, A_n)$ and $R_2(B_1, B_2, \dots, B_n)$ must have the same number of attributes, and the domains of corresponding attributes must be compatible; that is, $\text{dom}(A_i) = \text{dom}(B_i)$ for $i=1, 2, \dots, n$.
- The resulting relation for $R_1 \cup R_2$, $R_1 \cap R_2$, or $R_1 - R_2$ has the same attribute names as the *first* operand relation R_1 (by convention).

Relational Algebra Operations From Set Theory

● UNION Example

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	

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

STUDENT \cup INSTRUCTOR

Relational Algebra Operations From Set Theory (cont.) – use Fig. 6.4

Figure 7.11 Illustrating the set operations union, intersection, and difference. (a) Two union compatible relations.
(b) STUDENT \cup INSTRUCTOR. (c) STUDENT \cup INSTRUCTOR.
(d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a)	<table border="1"><thead><tr><th>STUDENT</th><th>FN</th><th>LN</th></tr></thead><tbody><tr><td>Susan</td><td>Yao</td><td></td></tr><tr><td>Ramesh</td><td>Shah</td><td></td></tr><tr><td>Johnny</td><td>Kohler</td><td></td></tr><tr><td>Barbara</td><td>Jones</td><td></td></tr><tr><td>Amy</td><td>Ford</td><td></td></tr><tr><td>Jimmy</td><td>Wang</td><td></td></tr><tr><td>Ernest</td><td>Gilbert</td><td></td></tr></tbody></table>	STUDENT	FN	LN	Susan	Yao		Ramesh	Shah		Johnny	Kohler		Barbara	Jones		Amy	Ford		Jimmy	Wang		Ernest	Gilbert		<table border="1"><thead><tr><th>INSTRUCTOR</th><th>FNAME</th><th>LNAME</th></tr></thead><tbody><tr><td>John</td><td>Smith</td><td></td></tr><tr><td>Ricardo</td><td>Browne</td><td></td></tr><tr><td>Susan</td><td>Yao</td><td></td></tr><tr><td>Francis</td><td>Johnson</td><td></td></tr><tr><td>Ramesh</td><td>Shah</td><td></td></tr></tbody></table>	INSTRUCTOR	FNAME	LNAME	John	Smith		Ricardo	Browne		Susan	Yao		Francis	Johnson		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																																											
(b)	<table border="1"><thead><tr><th>FN</th><th>LN</th></tr></thead><tbody><tr><td>Susan</td><td>Yao</td></tr><tr><td>Ramesh</td><td>Shah</td></tr><tr><td>Johnny</td><td>Kohler</td></tr><tr><td>Barbara</td><td>Jones</td></tr><tr><td>Amy</td><td>Ford</td></tr><tr><td>Jimmy</td><td>Wang</td></tr><tr><td>Ernest</td><td>Gilbert</td></tr><tr><td>John</td><td>Smith</td></tr><tr><td>Ricardo</td><td>Browne</td></tr><tr><td>Francis</td><td>Johnson</td></tr></tbody></table>	FN	LN	Susan	Yao	Ramesh	Shah	Johnny	Kohler	Barbara	Jones	Amy	Ford	Jimmy	Wang	Ernest	Gilbert	John	Smith	Ricardo	Browne	Francis	Johnson	<table border="1"><thead><tr><th>FN</th><th>LN</th></tr></thead><tbody><tr><td>Susan</td><td>Yao</td></tr><tr><td>Ramesh</td><td>Shah</td></tr></tbody></table>	FN	LN	Susan	Yao	Ramesh	Shah														
FN	LN																																											
Susan	Yao																																											
Ramesh	Shah																																											
Johnny	Kohler																																											
Barbara	Jones																																											
Amy	Ford																																											
Jimmy	Wang																																											
Ernest	Gilbert																																											
John	Smith																																											
Ricardo	Browne																																											
Francis	Johnson																																											
FN	LN																																											
Susan	Yao																																											
Ramesh	Shah																																											
(d)	<table border="1"><thead><tr><th>FN</th><th>LN</th></tr></thead><tbody><tr><td>Johnny</td><td>Kohler</td></tr><tr><td>Barbara</td><td>Jones</td></tr><tr><td>Amy</td><td>Ford</td></tr><tr><td>Jimmy</td><td>Wang</td></tr><tr><td>Ernest</td><td>Gilbert</td></tr></tbody></table>	FN	LN	Johnny	Kohler	Barbara	Jones	Amy	Ford	Jimmy	Wang	Ernest	Gilbert	<table border="1"><thead><tr><th>FNAME</th><th>LNAME</th></tr></thead><tbody><tr><td>John</td><td>Smith</td></tr><tr><td>Ricardo</td><td>Browne</td></tr><tr><td>Francis</td><td>Johnson</td></tr></tbody></table>	FNAME	LNAME	John	Smith	Ricardo	Browne	Francis	Johnson																						
FN	LN																																											
Johnny	Kohler																																											
Barbara	Jones																																											
Amy	Ford																																											
Jimmy	Wang																																											
Ernest	Gilbert																																											
FNAME	LNAME																																											
John	Smith																																											
Ricardo	Browne																																											
Francis	Johnson																																											

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Relational Algebra Operations From Set Theory (cont.)

● INTERSECTION OPERATION

The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S . The two operands must be "type compatible"

Example: The result of the intersection operation (figure below) includes only those who are both students and instructors.

FN	LN
Susan	Yao
Ramesh	Shah

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

STUDENT \cap INSTRUCTOR

Relational Algebra Operations From Set Theory (cont.)

● Set Difference (or MINUS) Operation

The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S . The two operands must be "type compatible".

Example: The figure shows the names of students who are not instructors, and the names of instructors who are not students.

STUDENT	FN	LN
Susan	Yao	
Ramesh	Shah	
Johnny	Kohler	
Barbara	Jones	
Amy	Ford	
Jimmy	Wang	
Ernest	Gilbert	

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

STUDENT-INSTRUCTOR

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

INSTRUCTOR-STUDENT

Relational Algebra Operations From Set Theory (cont.)

- Notice that 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 as both are *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

$$R - S \neq S - R$$

Relational Algebra Operations From Set Theory (cont.)

● CARTESIAN (or cross product) Operation

- This operation is used to combine tuples from two relations in a combinatorial fashion. 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.
- The two operands do NOT have to be "type compatible"

Example:

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

EMPNAMES $\leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE_EMPS)$

EMP_DEPENDENTS $\leftarrow EMPNAMES \times DEPENDENT$

Relational Algebra Operations From Set Theory (cont.)

Figure 7.12 An illustration of the CARTESIAN PRODUCT operation.

FEMALE_EMPS	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Alice	J	Zelena	888887777	1966-07-19	2021 Castle Spring,TX	F	25000	887654321	4	
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry Bellion,TX	F	43000	888885556	4	
Joyce	A	English	453453453	1972-07-31	5681 Rice Houston,TX	F	35000	333445556	6	

EMPNAME	FNAME	LNAME	SSN
Alice	Zelena	888887777	
Jennifer	Wallace	987654321	
Joyce	English	453453453	

EMP_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	***
Alice	Zelena	888887777	333445556	Alice	F	1966-04-05	***	
Alice	Zelena	888887777	333445556	Theodore	M	1963-10-25	***	
Alice	Zelena	888887777	333445556	Jay	F	1966-05-03	***	
Alice	Zelena	888887777	187654321	Abner	M	1942-02-28	***	
Alice	Zelena	888887777	123456789	Michael	M	1966-01-04	***	
Alice	Zelena	888887777	123456789	Alice	F	1966-12-30	***	
Alice	Zelena	888887777	123456789	Elizabeth	F	1967-05-05	***	
Jennifer	Wallace	987654321	333445556	Alice	F	1966-04-05	***	
Jennifer	Wallace	987654321	333445556	Theodore	M	1963-10-25	***	
Jennifer	Wallace	987654321	333445556	Jay	F	1966-05-03	***	
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	***	
Jennifer	Wallace	987654321	123456789	Michael	M	1966-01-04	***	
Jennifer	Wallace	987654321	123456789	Alice	F	1966-12-30	***	
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	***	
Joyce	English	453453453	333445556	Alice	F	1966-04-05	***	
Joyce	English	453453453	333445556	Theodore	M	1963-10-25	***	
Joyce	English	453453453	333445556	Jay	F	1966-05-03	***	
Joyce	English	453453453	187654321	Abner	M	1942-02-28	***	
Joyce	English	453453453	123456789	Michael	M	1966-01-04	***	
Joyce	English	453453453	123456789	Alice	F	1966-12-30	***	
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	

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Binary Relational Operations

● JOIN Operation

- The sequence of cartesian product followed by select is used quite commonly to identify and select related tuples from two relations, a special operation, called **JOIN**. It is denoted by a \bowtie
- This operation is very important for any relational database with more than a single relation, because 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:

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

where R and S can be any relations that result from general *relational algebra expressions*.

Binary Relational Operations (cont.)

Example: Suppose that we want to 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 MGRSSN value in the department tuple. We do this by using the join \bowtie operation.

DEPT_MGR \leftarrow DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

DEPT_MGR	DNAME	DNUMBER	MGRSSN	...	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	• • •

FIGURE 6.6 Result of the JOIN operation $DEPT_MGR \leftarrow DEPARTMENT \bowtie_{MGRSSN=SSN} EMPLOYEE$.

Binary Relational Operations (cont.)

- **EQUIJOIN Operation**

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 (whose names need not be identical) that have *identical values* in every tuple.

The JOIN seen in the previous example was EQUIJOIN.

- **NATURAL JOIN Operation**

Because one of each pair of attributes with identical values is superfluous, a new operation called natural join—denoted by $*$ —was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, have the **same name** in both relations. If this is not the case, a renaming operation is applied first.

Binary Relational Operations (cont.)

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

(a)

PROJ_DEPT	PNAME	PNUMBER	PLOCATION	DNUM	DNAME	MGRSSN	MGRSTARTDATE
ProductX	1	Bellaire	5	Research	333445555	1988-05-22	
ProductY	2	Sugarland	5	Research	333445555	1988-05-22	
ProductZ	3	Houston	5	Research	333445555	1988-05-22	
Computerization	10	Stafford	4	Administration	987654321	1995-01-01	
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19	
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01	

(b)

DEPT_LOCS	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	LOCATION
Headquarters	1	888665555	1981-06-19	Houston	
Administration	4	987654321	1995-01-01	Stafford	
Research	5	333445555	1988-05-22	Bellaire	
Research	5	333445555	1988-05-22	Sugarland	
Research	5	333445555	1988-05-22	Houston	

FIGURE 6.7 Results of two NATURAL JOIN operations. (a) PROJ_DEPT \leftarrow PROJECT * DEPT. (b) DEPT_LOCS \leftarrow DEPARTMENT * DEPT_LOCATIONS.

Complete Set of Relational Operations

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

- For example:

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

$$R \bowtie_{\text{join condition}} S = \sigma_{\text{join condition}}(R \times S)$$

Binary Relational Operations (cont.)

● DIVISION Operation

- The division operation is applied to two relations $R(Z) \div S(X)$, where X subset Z . Let $Y = Z - X$ (and hence $Z = X \cup Y$); that is, let Y be the set of attributes of R that are not attributes of S .
- 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 .
- For a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every* tuple in S .

Binary Relational Operations (cont.)

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

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

Recap of Relational Algebra Operations

TABLE 6.1 OPERATIONS OF RELATIONAL ALGEBRA

Operation	Purpose	Notation
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{\langle \text{SELECTION CONDITION} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{\langle \text{ATTRIBUTE LISTS} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\langle \text{JOIN CONDITION} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{JOIN CONDITION} \rangle} R_2$, OR $R_1 \bowtie_{\langle \text{JOIN ATTRIBUTES } 1 \rangle},$ $\quad \quad \quad \langle \text{JOIN ATTRIBUTES } 2 \rangle} R_2$
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.	$R_1 \bowtie_{\langle \text{JOIN CONDITIONS} \rangle} R_2$, OR $R_1 \bowtie_{\langle \text{JOIN ATTRIBUTES } 1 \rangle},$ $\quad \quad \quad \langle \text{JOIN ATTRIBUTES } 2 \rangle} R_2$ 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 $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Additional Relational Operations

● Aggregate Functions and Grouping

- A type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database.
- Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are 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.

Additional Relational Operations (cont.)

(a)

R	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

Additional Relational Operations (cont.)

Use of the Functional operator \mathcal{F}

$\mathcal{F}_{\text{MAX } \text{Salary}}$ (**Employee**) retrieves the maximum salary value from the Employee relation

$\mathcal{F}_{\text{MIN } \text{Salary}}$ (**Employee**) retrieves the minimum Salary value from the Employee relation

$\mathcal{F}_{\text{SUM } \text{Salary}}$ (**Employee**) retrieves the sum of the Salary from the Employee relation

DNO $\mathcal{F}_{\text{COUNT } \text{SSN}, \text{AVERAGE } \text{Salary}}$ (**Employee**) groups employees by DNO (department number) and computes the count of employees and average salary per department. [Note: count just counts the number of rows, without removing duplicates]

Additional Relational Operations (cont.)

● Recursive Closure Operations

- Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure**. This operation is applied to a **recursive relationship**.
- An example of a recursive operation is to retrieve all SUPERVISEES of an EMPLOYEE e at all levels—that is, all EMPLOYEE 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 .
- Although it is possible to retrieve employees at each level and then take their union, we cannot, in general, specify a query such as “retrieve the supervisees of ‘James Borg’ at all levels” without utilizing a looping mechanism.
- The SQL3 standard includes syntax for recursive closure.

Additional Relational Operations (cont.)

(Borg's SSN is 888665555)

(SSN) (SUPERSSN)

SUPERVISION	SSN1	SSN2
	123456789	333445555
	333445555	888665555
	999887777	987654321
	987654321	888665555
	666884444	333445555
	453453453	333445555
	987987987	987654321

RESULT 1	SSN
	333445555
	987654321

(Supervised by Borg)

RESULT 2	SSN
	123456789
	999887777
	666884444
	453453453
	987987987

(Supervised by Borg's subordinates)

RESULT	SSN
	123456789
	999887777
	666884444
	453453453
	987987987
	333445555
	987654321

(RESULT1 \cup RESULT2)

Additional Relational Operations (cont.)

● The OUTER JOIN Operation

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

Additional Relational Operations (cont.)

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

Additional Relational Operations (cont.)

● OUTER UNION Operations

- The outer union operation was developed to take the union of tuples from two relations if the relations are *not union 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 $T(X, Y, Z)$.
- **Example:** An outer union can be applied to two relations whose schemas are STUDENT(Name, SSN, Department, Advisor) and 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. If a student is also an instructor, both Advisor and Rank will have a value; otherwise, one of these two attributes will be null.

The result relation STUDENT_OR_INSTRUCTOR will have the following attributes:

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

Examples of Queries in Relational Algebra

- **Q1: Retrieve the name and address of all employees who work for the ‘Research’ department.**

$\text{RESEARCH_DEPT} \leftarrow \sigma_{\text{DNAME}=\text{'Research'}}(\text{DEPARTMENT})$

$\text{RESEARCH_EMPS} \leftarrow (\text{RESEARCH_DEPT} \bowtie_{\text{DNUMBER}=\\ \text{DNOEMPLOYEE}} \text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{ADDRESS}}(\text{RESEARCH_EMPS})$

- **Q6: Retrieve the names of employees who have no dependents.**

$\text{ALL_EMPS} \leftarrow \pi_{\text{SSN}}(\text{EMPLOYEE})$

$\text{EMPS_WITH_DEPS(SSN)} \leftarrow \pi_{\text{ESSN}}(\text{DEPENDENT})$

$\text{EMPS_WITHOUT_DEPS} \leftarrow (\text{ALL_EMPS} - \text{EMPS_WITH_DEPS})$

$\text{RESULT} \leftarrow \pi_{\text{LNAME}, \text{FNAME}}(\text{EMPS_WITHOUT_DEPS} * \text{EMPLOYEE})$

Relational Calculus

- A **relational calculus** expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in **tuple calculus**) or over columns of the stored relations (in **domain calculus**).
- In a calculus expression, there is *no order of operations* to specify how to retrieve the query result—a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus.
- Relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence relational algebra can be considered as a **procedural** way of stating a query.

Tuple Relational Calculus

- The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation.
- A simple tuple relational calculus query is of the form
 $\{t \mid \text{COND}(t)\}$
where t is a tuple variable and COND (t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND (t).

Example: To find the first and last names of all employees whose salary is above \$50,000, we can write the following tuple calculus expression:

{t.FNAME, t.LNAME | EMPLOYEE(t) AND t.SALARY>50000}

The condition EMPLOYEE(t) specifies that the **range relation** of tuple variable t is EMPLOYEE. The first and last name (PROJECTION $\pi_{\text{FNAME, LNAME}}$) of each EMPLOYEE tuple t that satisfies the condition $t.\text{SALARY}>50000$ (SELECTION $\sigma_{\text{SALARY} > 50000}$) will be retrieved.

The Existential and Universal Quantifiers

- Two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** (\forall) and the **existential quantifier** (\exists).
- Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an $(\forall t)$ or $(\exists t)$ clause; otherwise, it is **free**.
- If F is a formula, then so is $(\exists t)(F)$, where t is a tuple variable. The formula $(\exists t)(F)$ is true if the formula F evaluates to true for *some* (at least one) tuple assigned to free occurrences of t in F ; otherwise $(\exists t)(F)$ is **false**.
- If F is a formula, then so is $(\forall t)(F)$, where t is a tuple variable. The formula $(\forall t)(F)$ is true if the formula F evaluates to true for *every tuple* (in the universe) assigned to free occurrences of t in F ; otherwise $(\forall t)(F)$ is **false**. It is called the universal or “for all” quantifier because every tuple in “the universe of” tuples must make F true to make the quantified formula true.

Example Query Using Existential Quantifier

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

Query :

$$\{t.\text{FNAME}, t.\text{LNAME}, t.\text{ADDRESS} \mid \text{EMPLOYEE}(t) \text{ and } (\exists d) \\ (\text{DEPARTMENT}(d) \text{ and } d.\text{DNAME} = \text{'Research'} \text{ and } d.\text{DNUMBER} = t.\text{DNO}) \}$$

- The *only free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (|). In above query, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified in the query, the attributes FNAME, LNAME, and ADDRESS are retrieved for each such tuple.
- The conditions EMPLOYEE (t) and DEPARTMENT(d) specify the range relations for t and d. The condition d.DNAME = ‘Research’ is a selection condition and corresponds to a SELECT operation in the relational algebra, whereas the condition d.DNUMBER = t.DNO is a JOIN condition.

Example Query Using Universal Quantifier

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

Query :

```
{e.LNAME, e.FNAME | EMPLOYEE(e) and ( (forall x)(not(PROJECT(x)) or  
not(x.DNUM=5)  
OR ( (exists w)(WORKS_ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO) ) ) )}
```

- Exclude from the universal quantification all tuples that we are not interested in by making the condition true *for all such tuples*. The first tuples to exclude (by making them evaluate automatically to true) are those that are not in the relation R of interest.
- In query above, using the expression not(PROJECT(x)) inside the universally quantified formula evaluates to true all tuples x that are not in the PROJECT relation. Then we exclude the tuples we are not interested in from R itself. The expression not(x.DNUM=5) evaluates to true all tuples x that are in the project relation but are not controlled by department 5.
- Finally, we specify a condition that must hold on all the remaining tuples in R.
 $((exists w)(WORKS_ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO)$

Languages Based on Tuple Relational Calculus

- The language **SQL** is based on tuple calculus. It uses the basic
SELECT <list of attributes>
FROM <list of relations>
WHERE <conditions>
block structure to express the queries in tuple calculus where the SELECT clause mentions the attributes being projected, the FROM clause mentions the relations needed in the query, and the WHERE clause mentions the selection as well as the join conditions.
SQL syntax is expanded further to accommodate other operations. (See Chapter 8).
- Another language which is based on tuple calculus is **QUEL** which actually uses the range variables as in tuple calculus.
Its syntax includes:
RANGE OF <variable name> IS <relation name>
Then it uses
RETRIEVE <list of attributes from range variables>
WHERE <conditions>
This language was proposed in the relational DBMS INGRES.

The Domain Relational Calculus

- Another variation of relational calculus called the domain relational calculus, or simply, **domain calculus** is equivalent to tuple calculus and to relational algebra.
- The language called QBE (Query-By-Example) that is related to domain calculus was developed almost concurrently to SQL at IBM Research, Yorktown Heights, New York. Domain calculus was thought of as a way to explain what QBE does.
- Domain calculus differs from tuple calculus in the *type of variables* used in formulas: rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute.
- An expression of the domain calculus is of the form
$$\{x_1, x_2, \dots, x_n \mid \text{COND}(x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m})\}$$
where $x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}$ are domain variables that range over domains (of attributes) and COND is a **condition** or **formula** of the domain relational calculus.

Example Query Using Domain Calculus

- Retrieve the birthdate and address of the employee whose name is ‘John B. Smith’.

Query :

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

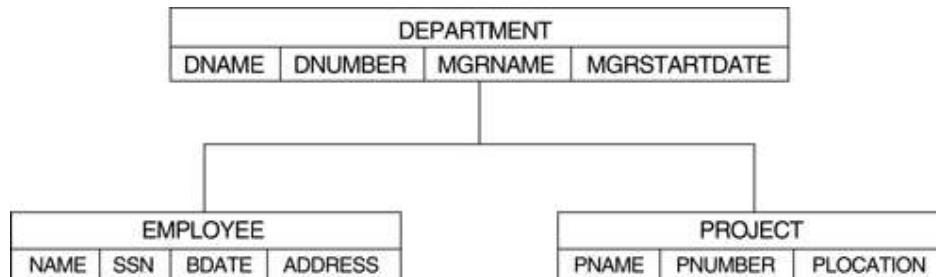
- Ten variables for the employee relation are needed, one to range over the domain of each attribute in order. Of the ten variables q, r, s, \dots, z , only u and v are free.
- Specify the *requested attributes*, BDATE and ADDRESS, by the free domain variables u for BDATE and v for ADDRESS.
- Specify the condition for selecting a tuple following the bar (\mid)—namely, that the sequence of values assigned to the variables $qrstuvwxyz$ be a tuple of the employee relation and that the values for q (FNAME), r (MINIT), and s (LNAME) be ‘John’, ‘B’, and ‘Smith’, respectively.

QBE: A Query Language Based on Domain Calculus (Appendix D)

- This language is based on the idea of giving an example of a query using **example elements**.
- An example element stands for a domain variable and is specified as an example value preceded by the underscore character.
- P. (called **P dot**) operator (for “print”) is placed in those columns which are requested for the result of the query.
- A user may initially start giving actual values as examples, but later can get used to providing a minimum number of variables as example elements.
- The language is very user-friendly, because it uses minimal syntax.
- QBE was fully developed further with facilities for grouping, aggregation, updating etc. and is shown to be equivalent to SQL.
- The language is available under QMF (Query Management Facility) of DB2 of IBM and has been used in various ways by other products like ACCESS of Microsoft, PARADOX.
- For details, see Appendix D in the text.

QBE Examples

- QBE initially presents a relational schema as a “blank schema” in which the user fills in the query as an example:



QBE Examples

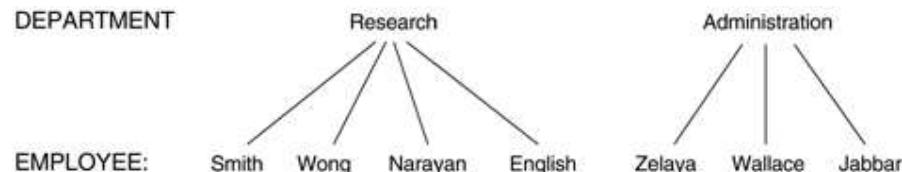
- The following domain calculus query can be successively minimized by the user as shown:

Query :

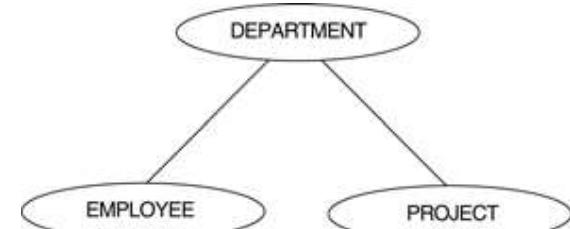
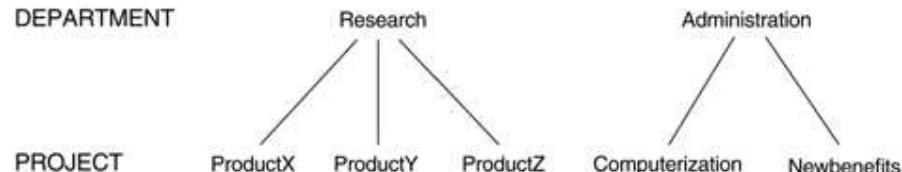
$$\{uv \mid (\exists q) (\exists r) (\exists s) (\exists t) (\exists w) (\exists x) (\exists y) (\exists z)$$

(EMPLOYEE(qrstuvwxyz) and q='John' and r='B' and s='Smith')

(a) DEPARTMENT



(b) DEPARTMENT



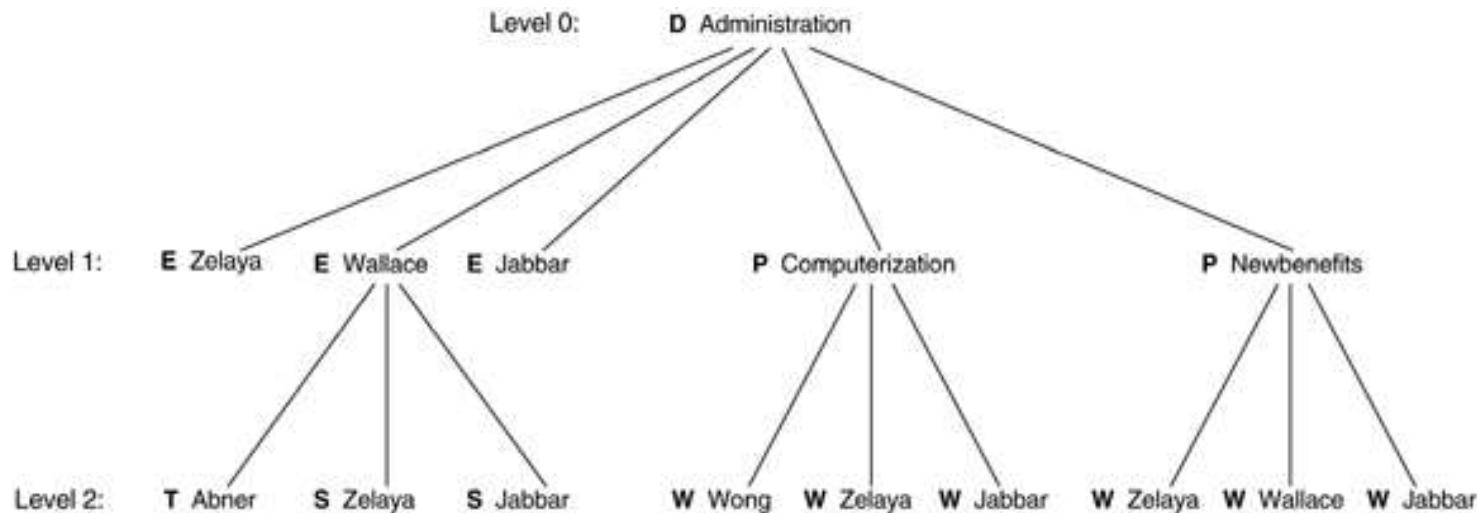
QBE Examples

Specifying complex conditions in QBE:

- A technique called the “condition box” is used in QBE to state more involved Boolean expressions as conditions.
- The D.4(a) gives employees who work on either project 1 or 2, whereas the query in D.4(b) gives those who work on both the projects.

QBE Examples

- Illustrating join in QBE. The join is simple accomplished by using the same example element in the columns being joined. Note that the Result is set us as an independent table.



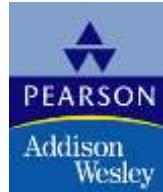
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 7

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



Chapter Outline

● ER-to-Relational Mapping Algorithm

- Step 1: Mapping of Regular Entity Types
- Step 2: Mapping of Weak Entity Types
- Step 3: Mapping of Binary 1:1 Relation Types
- Step 4: Mapping of Binary 1:N Relationship Types.
- Step 5: Mapping of Binary M:N Relationship Types.
- Step 6: Mapping of Multivalued attributes.
- Step 7: Mapping of N-ary Relationship Types.

● Mapping EER Model Constructs to Relations

- Step 8: Options for Mapping Specialization or Generalization.
- Step 9: Mapping of Union Types (Categories).

ER-to-Relational Mapping Algorithm

● Step 1: Mapping of Regular Entity Types.

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

Example: We create the relations EMPLOYEE, DEPARTMENT, and PROJECT in the relational schema corresponding to the regular entities in the ER diagram. SSN, DNUMBER, and PNUMBER are the primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT as shown.

FIGURE 7.1

The ER conceptual schema diagram for the COMPANY database.

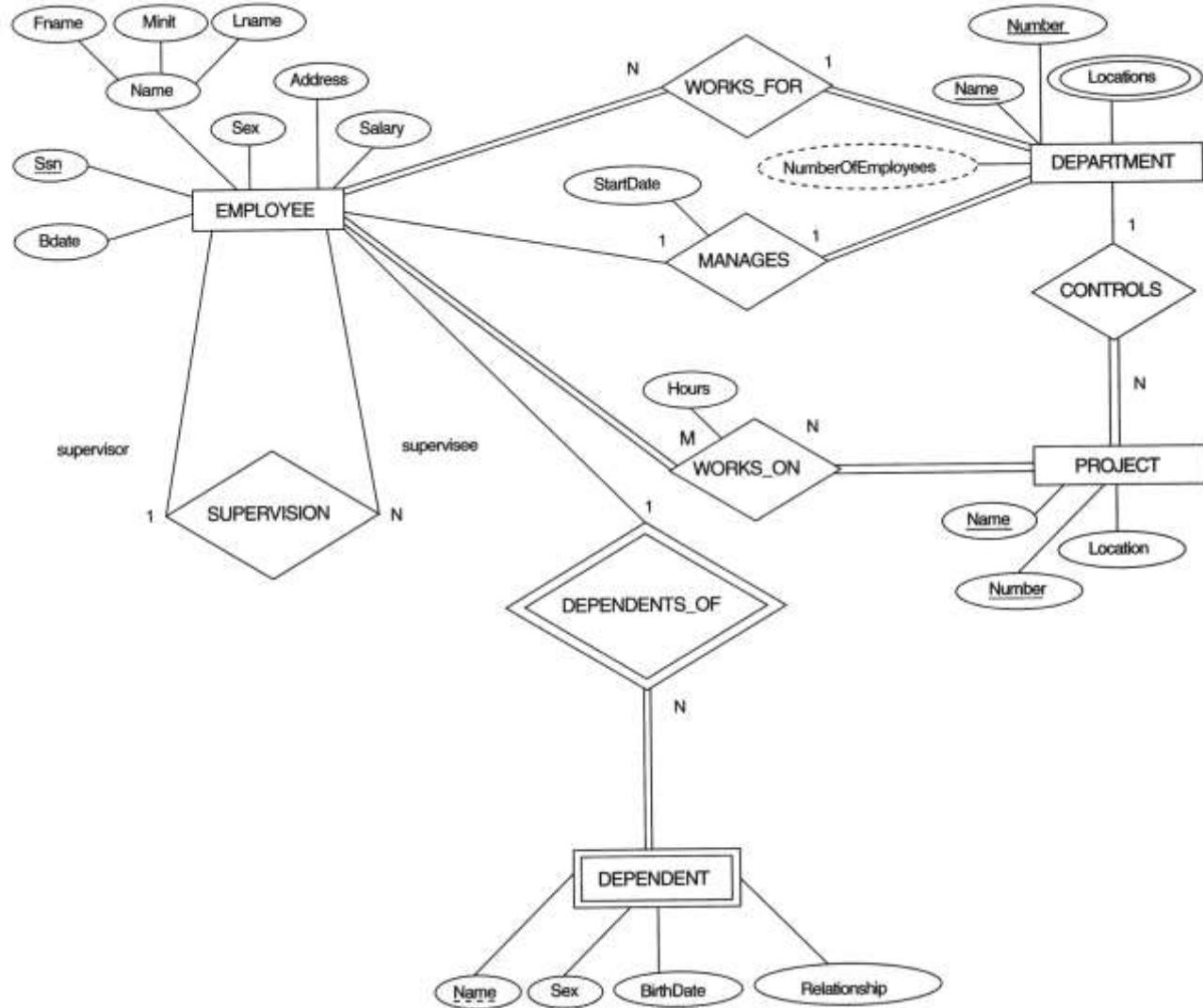
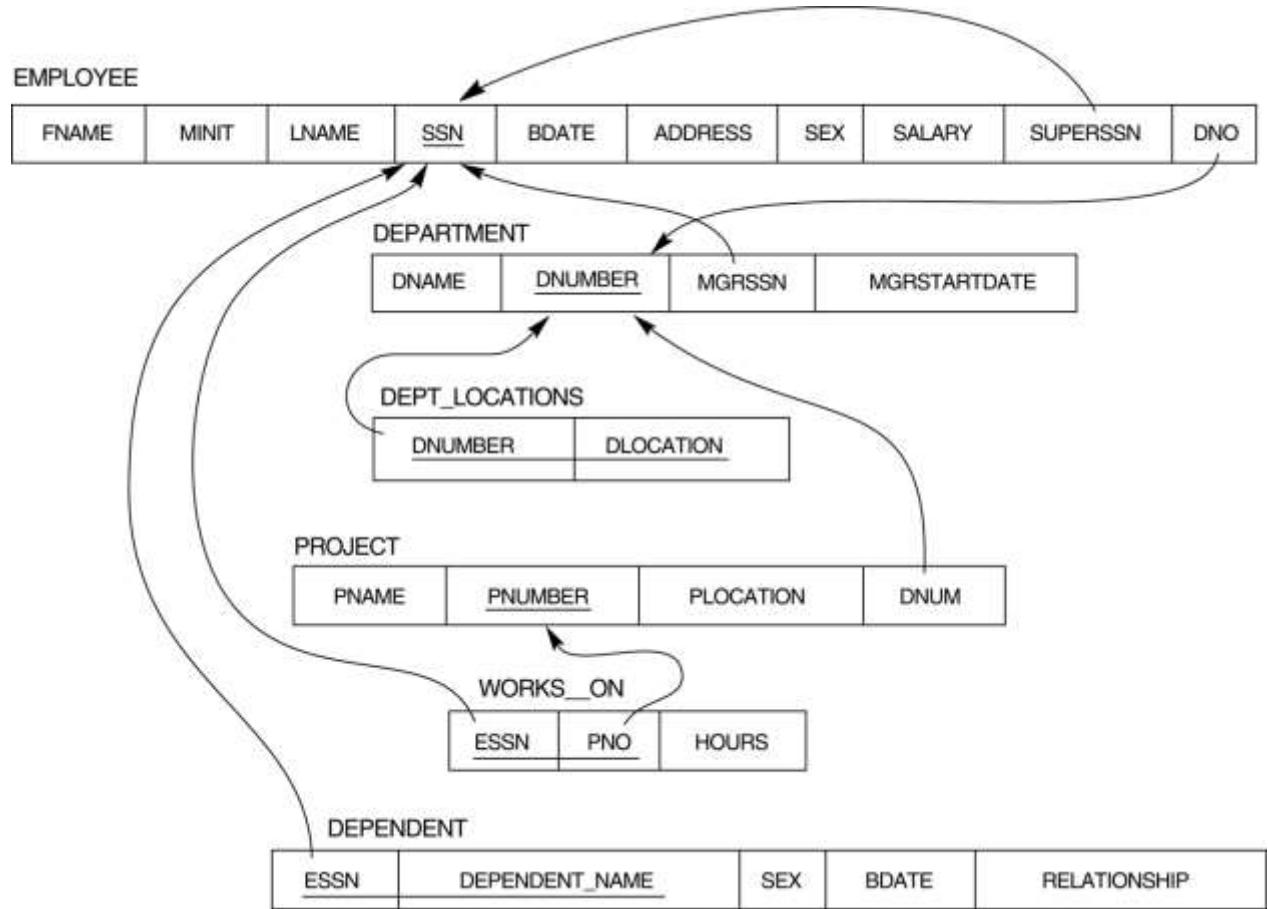


FIGURE 7.2

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



ER-to-Relational Mapping Algorithm (cont)

● Step 2: Mapping of Weak Entity Types

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

Example: Create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. Include the primary key SSN of the EMPLOYEE relation as a foreign key attribute of DEPENDENT (renamed to ESSN).

The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT_NAME} because DEPENDENT_NAME is the partial key of DEPENDENT.

ER-to-Relational Mapping Algorithm (cont)

● Step 3: Mapping of Binary 1:1 Relation Types

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

- (1) Foreign Key approach: Choose one of the relations-S, say-and include a foreign key in S the primary key of T. It is better to choose an entity type with *total participation* in R in the role of S. **Example**: 1:1 relation MANAGES is mapped by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total.
- (2) Merged relation option: An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when *both participations are total*.
- (3) Cross-reference or relationship relation option: The third alternative is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types.

ER-to-Relational Mapping Algorithm (cont)

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

- For each regular binary 1:N relationship type R, identify the relation S that represent 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 of the 1:N relation type as attributes of S.

Example: 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION in the figure. For WORKS_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO.

ER-to-Relational Mapping Algorithm (cont)

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

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

Example: The M:N relationship type WORKS_ON from the ER diagram is mapped by creating a relation WORKS_ON in the relational database schema. The primary keys of the PROJECT and EMPLOYEE relations are included as foreign keys in WORKS_ON and renamed PNO and ESSN, respectively.

Attribute HOURS in WORKS_ON represents the HOURS attribute of the relation type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {ESSN, PNO}.

ER-to-Relational Mapping Algorithm (cont)

● Step 6: Mapping of Multivalued attributes.

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

Example: The relation DEPT_LOCATIONS is created. 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 R is the combination of {DNUMBER, DLOCATION}.

ER-to-Relational Mapping Algorithm (cont)

● Step 7: Mapping of N-ary Relationship Types.

- For each n-ary relationship type R, where $n > 2$, create a new relationship S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types.
- Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S.

Example: The relationship type SUPPY in the ER below. This can be mapped to the relation SUPPLY shown in the relational schema, whose primary key is the combination of the three foreign keys {SNAME, PARTNO, PROJNAME}

FIGURE 4.11

Ternary relationship types. (a) The SUPPLY relationship.

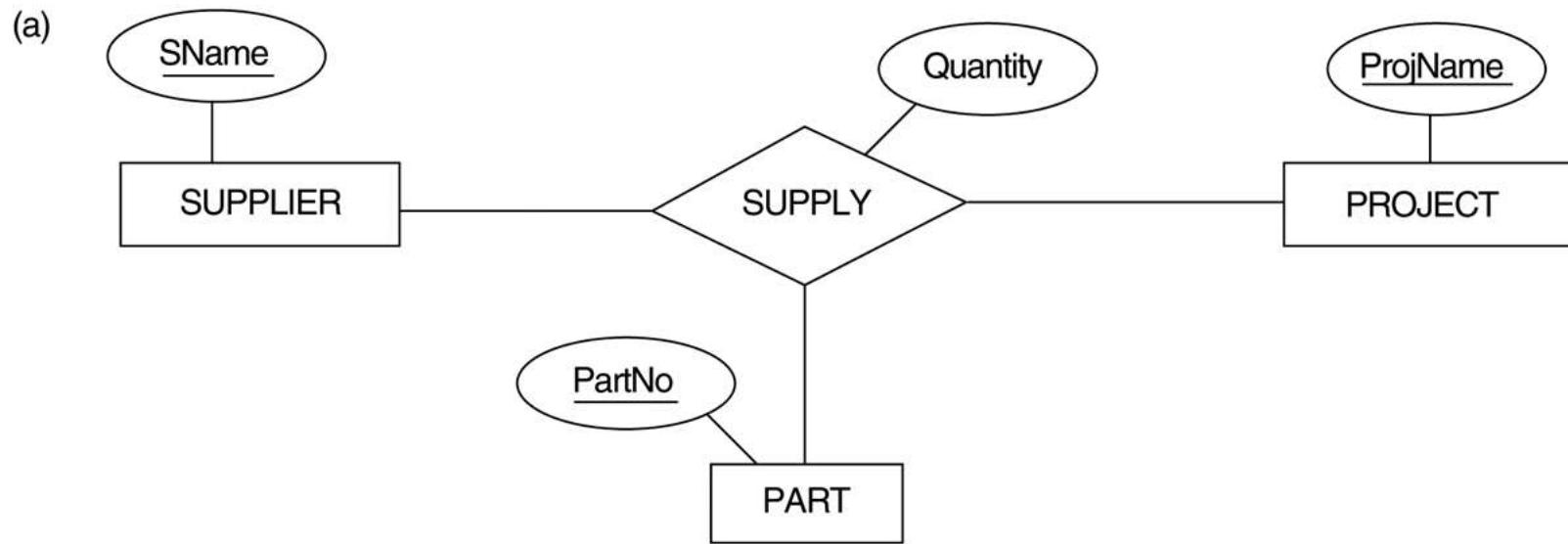


FIGURE 7.3

Mapping the *n*-ary relationship type SUPPLY from Figure 4.11a.

SUPPLIER

<u>SNAME</u>	• • •
--------------	-------

PROJECT

<u>PROJNAME</u>	• • •
-----------------	-------

PART

<u>PARTNO</u>	• • •
---------------	-------

SUPPLY

<u>SNAME</u>	<u>PROJNAME</u>	<u>PARTNO</u>	QUANTITY
--------------	-----------------	---------------	----------

Summary of Mapping constructs and constraints

Table 7.1 Correspondence between ER and Relational Models

ER Model	Relational Model
Entity type	“Entity” relation
1:1 or 1:N relationship type	Foreign key (or “relationship” relation)
M:N relationship type	“Relationship” relation and two foreign keys
<i>n</i> -ary relationship type	“Relationship” 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

Mapping EER Model Constructs to Relations

- **Step8: Options for Mapping Specialization or Generalization.**

Convert each specialization with m subclasses $\{S_1, S_2, \dots, S_m\}$ and generalized superclass C, where the attributes of C are $\{k, a_1, \dots, a_n\}$ and k is the (primary) key, into relational schemas using one of the four following options:

Option 8A: Multiple relations-Superclass and subclasses.

Create a relation L for C with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$ and $\text{PK}(L) = k$. Create a relation L_i for each subclass S_i , $1 < i < m$, with the attributes $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $\text{PK}(L_i) = k$. This option works **for any specialization** (total or partial, disjoint or overlapping).

Option 8B: Multiple relations-Subclass relations only

Create a relation L_i for each subclass S_i , $1 < i < m$, with the attributes $\text{Attr}(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \dots, a_n\}$ and $\text{PK}(L_i) = k$. This option only works for a specialization whose subclasses are **total** (every entity in the superclass must belong to (at least) one of the subclasses).

FIGURE 4.4
EER diagram
notation for an
attribute-
defined
specialization
on JobType.

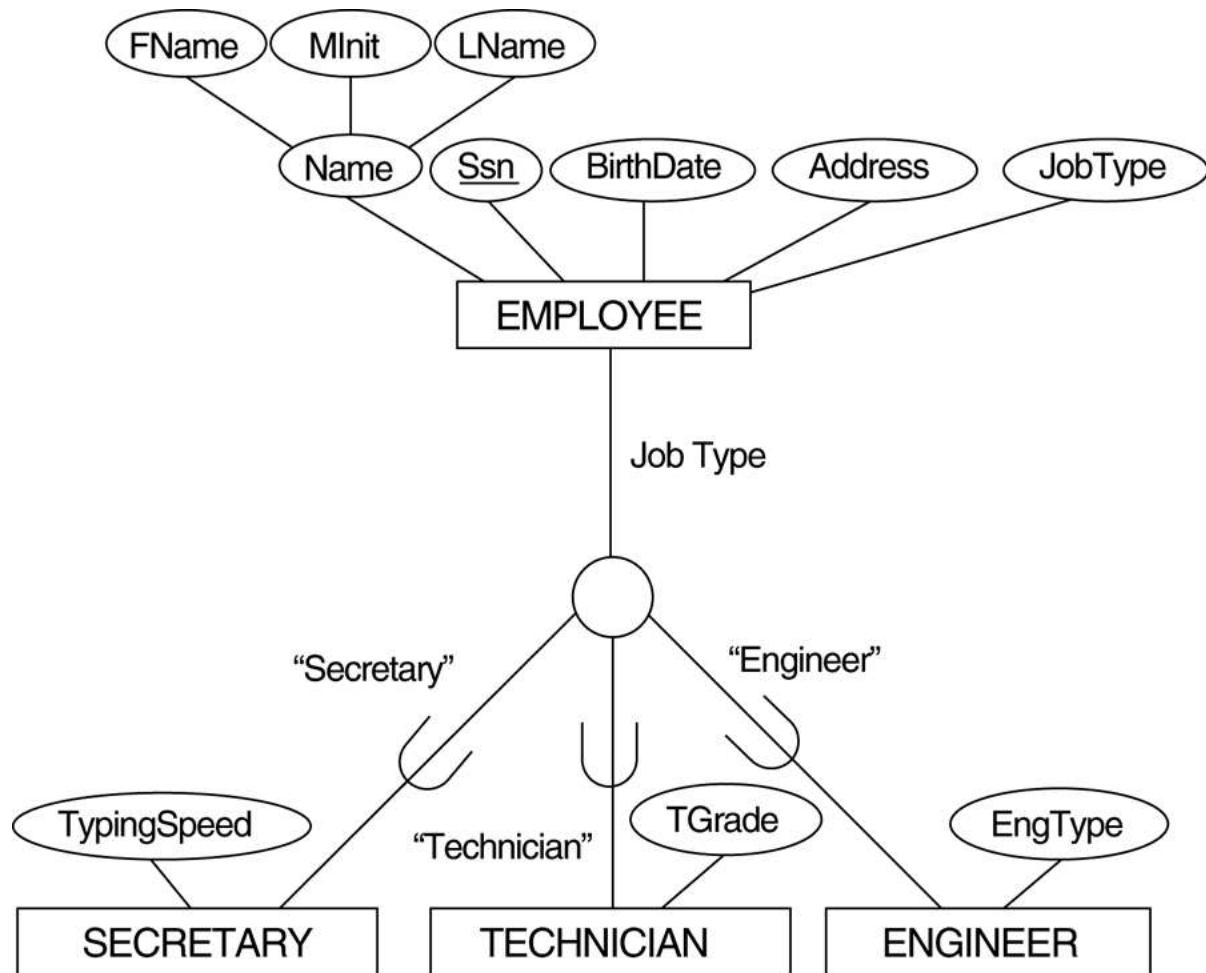


FIGURE 7.4

Options for mapping specialization or generalization.

(a) Mapping the EER schema in Figure 4.4 using option 8A.

(a)

EMPLOYEE

<u>SSN</u>	FName	MInit	LName	BirthDate	Address	JobType
------------	-------	-------	-------	-----------	---------	---------

SECRETARY

<u>SSN</u>	TypingSpeed
------------	-------------

TECHNICIAN

<u>SSN</u>	TGrade
------------	--------

ENGINEER

<u>SSN</u>	EngType
------------	---------

FIGURE 4.3

Generalization. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

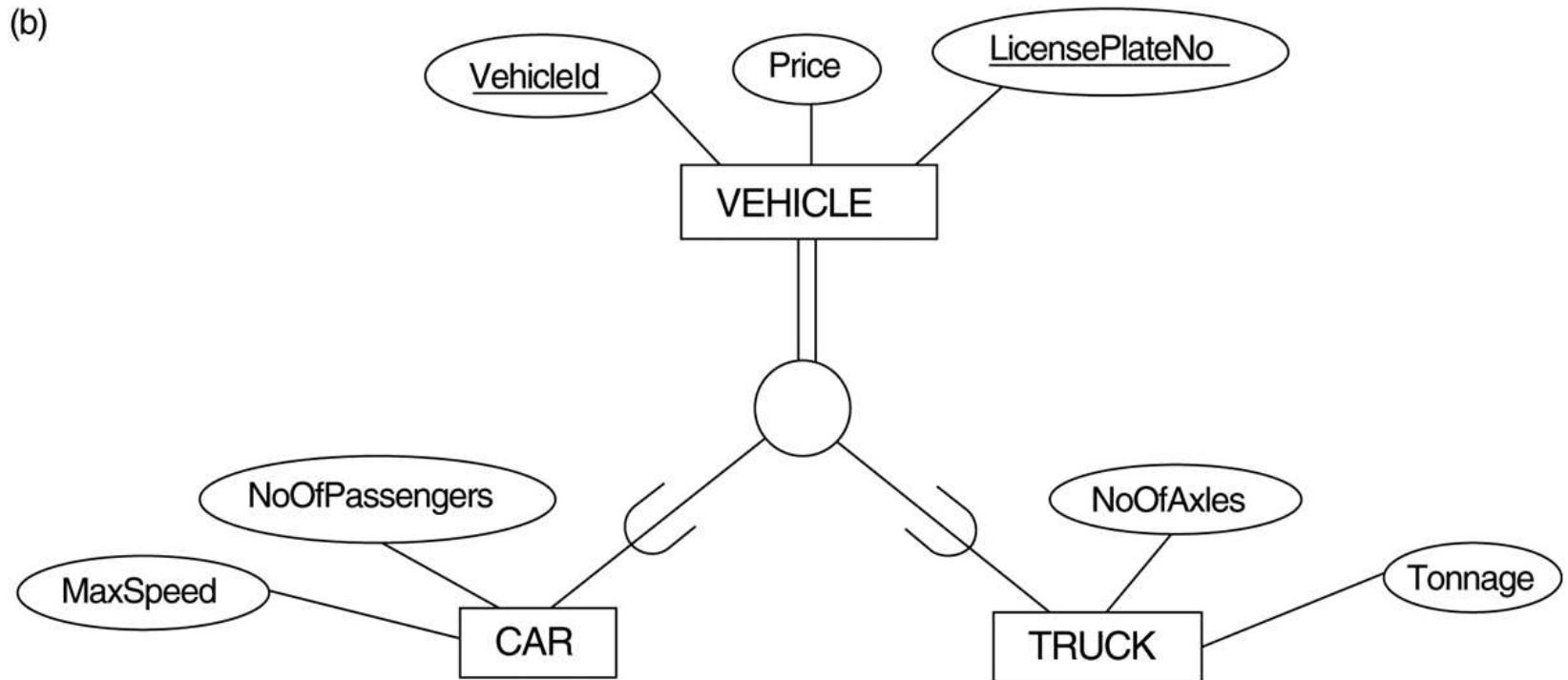


FIGURE 7.4

Options for mapping specialization or generalization.

(b) Mapping the EER schema in Figure 4.3b using option 8B.

(b) CAR

<u>VehicleId</u>	LicensePlateNo	Price	MaxSpeed	NoOfPassengers
------------------	----------------	-------	----------	----------------

TRUCK

<u>VehicleId</u>	LicensePlateNo	Price	NoOfAxles	
------------------	----------------	-------	-----------	--

Mapping EER Model Constructs to Relations (cont)

Option 8C: Single relation with one type attribute.

Create a single relation L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t\}$ and $\text{PK}(L) = k$. The attribute t is called a type (or **discriminating**) attribute that indicates the subclass to which each tuple belongs

Option 8D: Single relation with multiple type attributes.

Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \dots, t_m\}$ and $\text{PK}(L) = k$. Each t_i , $1 < i < m$, is a Boolean type attribute indicating whether a tuple belongs to the subclass S_i .

FIGURE 4.4
EER diagram
notation for an
attribute-
defined
specialization
on JobType.

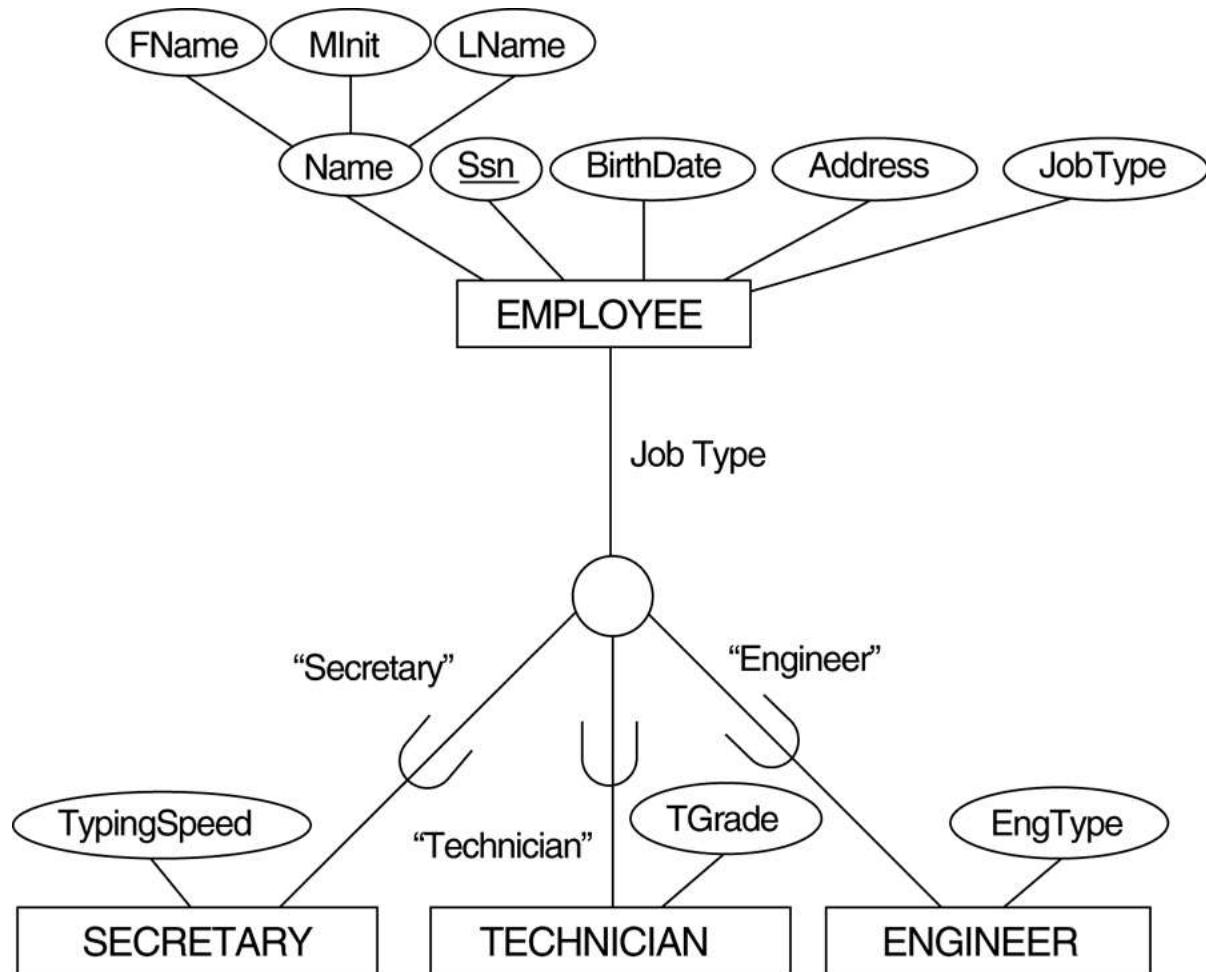


FIGURE 7.4

Options for mapping specialization or generalization.

(c) Mapping the EER schema in Figure 4.4 using option 8C.

(c) EMPLOYEE

SSN	FName	MInit	LName	BirthDate	Address	JobType	TypingSpeed	TGrade	
-----	-------	-------	-------	-----------	---------	---------	-------------	--------	--

FIGURE 4.5

EER diagram notation for an overlapping (nondisjoint) specialization.

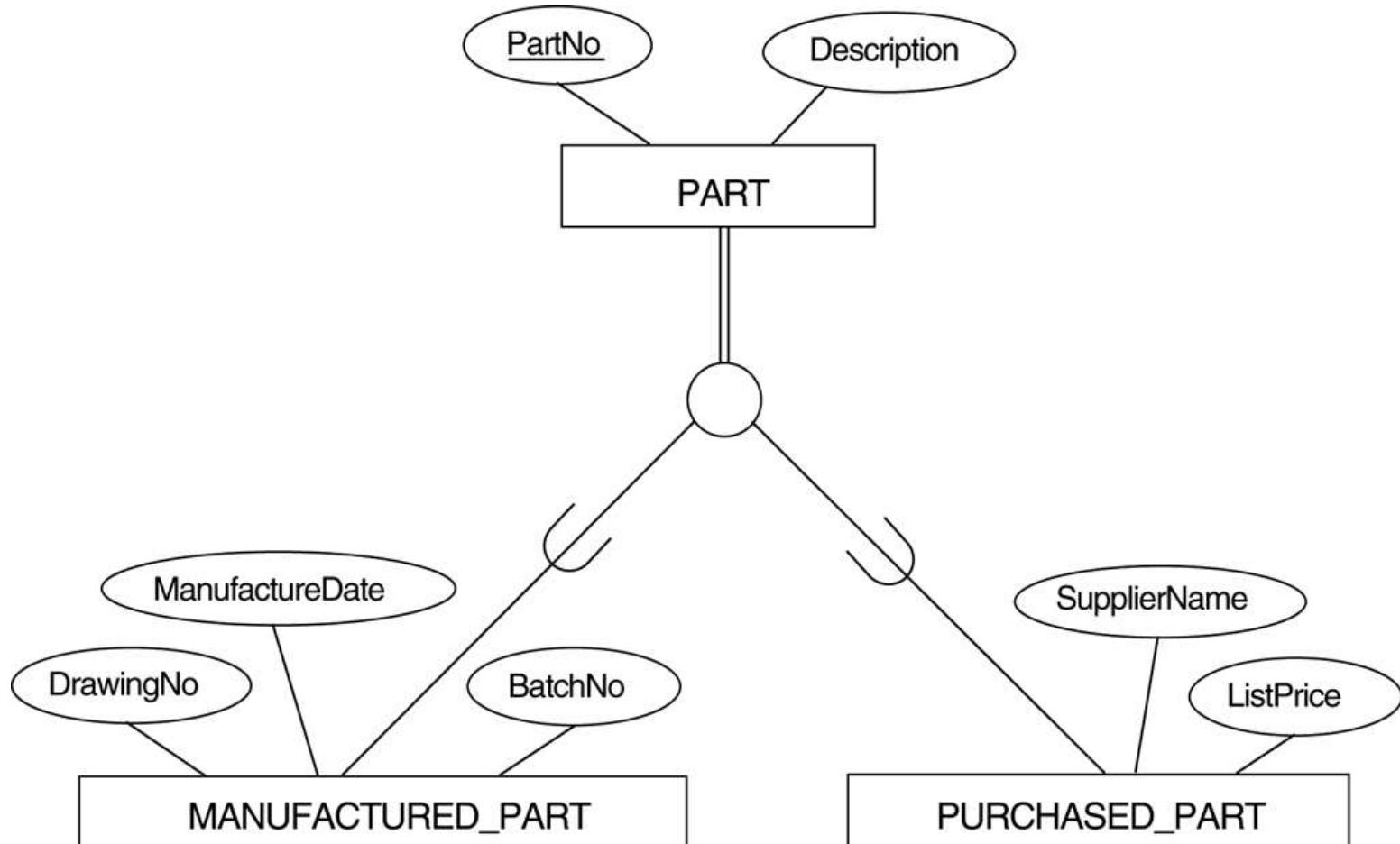


FIGURE 7.4

Options for mapping specialization or generalization.

(d) Mapping Figure 4.5 using option 8D with Boolean type fields Mflag and Pflag.

(d) PART

PartNo	Description	MFlag	DrawingNo	ManufactureDate	BatchNo	PFlag	SupplierName	ListPrice
--------	-------------	-------	-----------	-----------------	---------	-------	--------------	-----------

Mapping EER Model Constructs to Relations (cont)

- **Mapping of Shared Subclasses (Multiple Inheritance)**

A shared subclass, such as STUDENT_ASSISTANT, is a subclass of several classes, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category.

We can apply any of the options discussed in Step 8 to a shared subclass, subject to the restriction discussed in Step 8 of the mapping algorithm. Below both 8C and 8D are used for the shared class STUDENT_ASSISTANT.

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

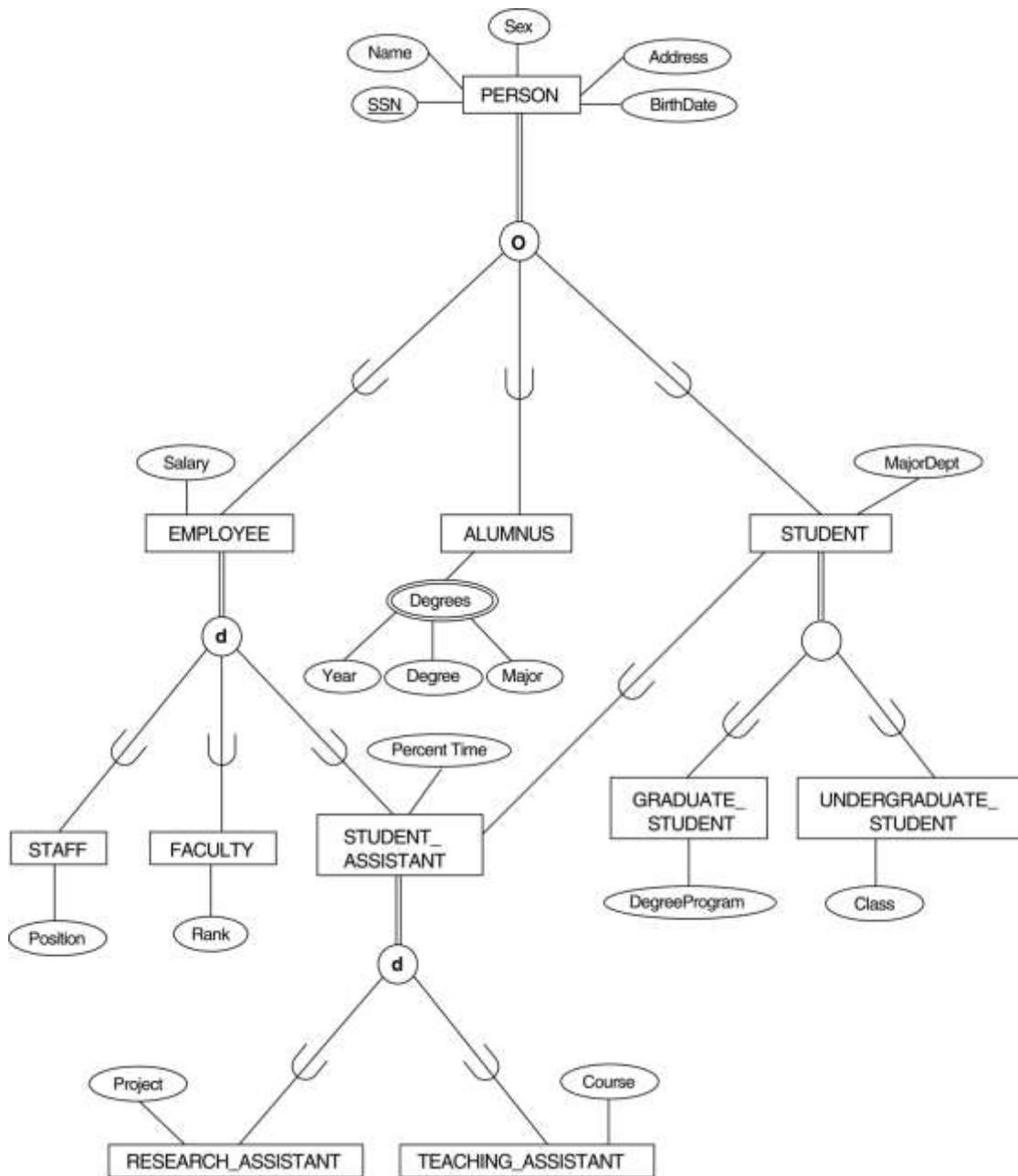


FIGURE 7.5

Mapping the EER specialization lattice in Figure 4.6 using multiple options.

PERSON

<u>SSN</u>	Name	BirthDate	Sex	Address
------------	------	-----------	-----	---------

EMPLOYEE

<u>SSN</u>	Salary	EmployeeType	Position	Rank	PercentTime	RAFlag	TAFlag	Project	
------------	--------	--------------	----------	------	-------------	--------	--------	---------	--

ALUMNUS

<u>SSN</u>	<u>SSN</u>	Year	Degree	
------------	------------	------	--------	--

ALUMNUS_DEGREES

STUDENT

<u>SSN</u>	MajorDept	GradFlag	UndergradFlag	DegreeProgram	Class	StudAssistFlag
------------	-----------	----------	---------------	---------------	-------	----------------

Mapping EER Model Constructs to Relations (cont)

- **Step 9: Mapping of Union Types (Categories).**

- For mapping a category whose defining superclass have different keys, it is customary to specify a new key attribute, called a **surrogate key**, when creating a relation to correspond to the category.
- In the example below we can create a relation OWNER to correspond to the OWNER category and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called OwnerId.

FIGURE 4.8

Two categories (union types): OWNER and REGISTERED_VEHICLE.

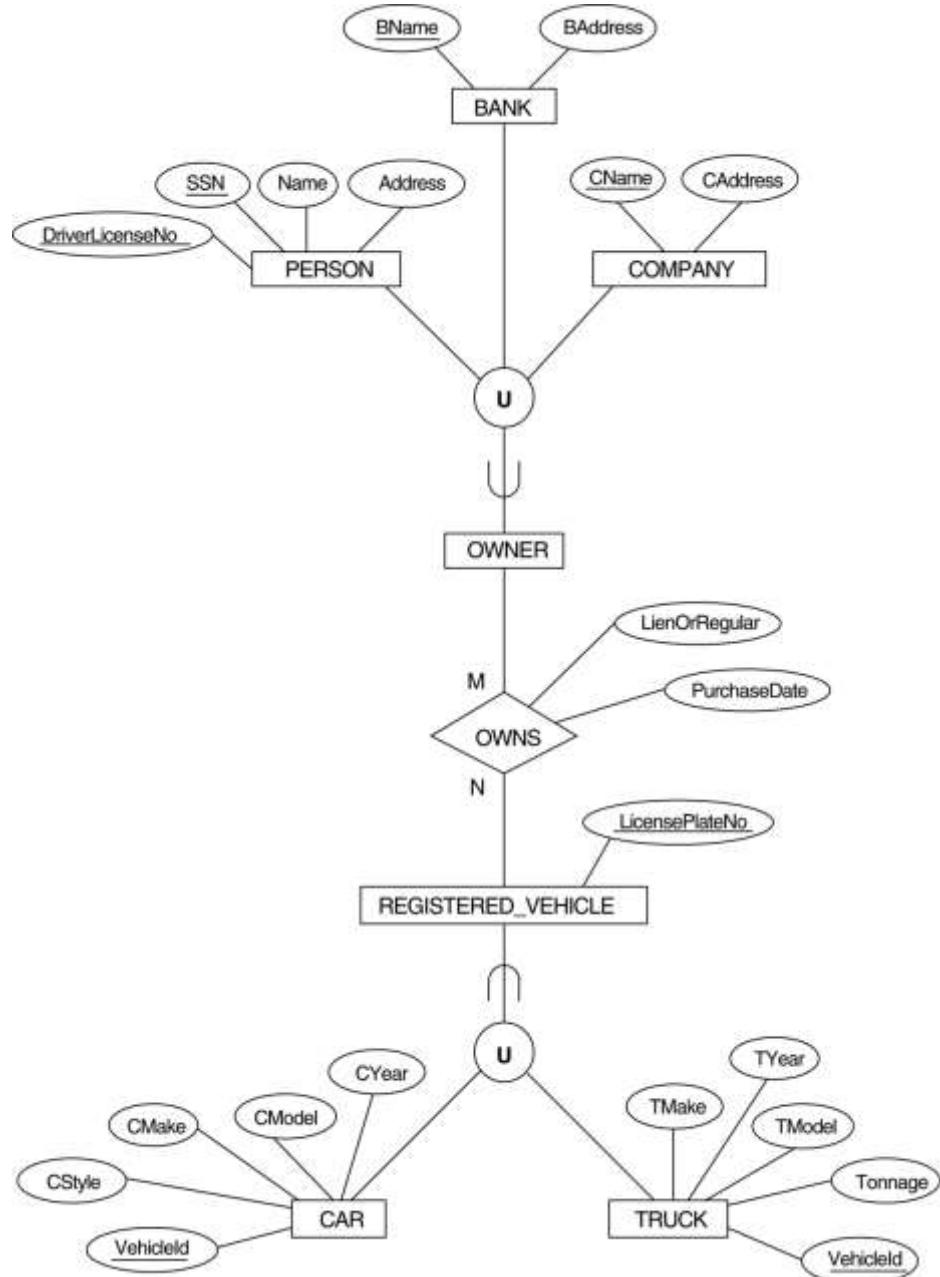


FIGURE 7.6

Mapping the EER categories (union types) in Figure 4.7 to relations.

PERSON				
<u>SSN</u>	DriverLicenseNo	Name	Address	
BANK				
<u>BName</u>	BAddress	OwnerId		
COMPANY				
<u>CName</u>	CAddress	OwnerId		
OWNER				
<u>OwnerId</u>				
REGISTERED_VEHICLE				
<u>VehicleId</u>	LicensePlateNumber			
CAR				
<u>VehicleId</u>	CStyle	CMake	CModel	
TRUCK				
<u>VehicleId</u>	TMake	TModel	Tonnage	TYear
OWNS				
<u>OwnerId</u>	<u>VehicleId</u>	PurchaseDate	LienOrRegular	

Mapping Exercise

Exercise 7.4.

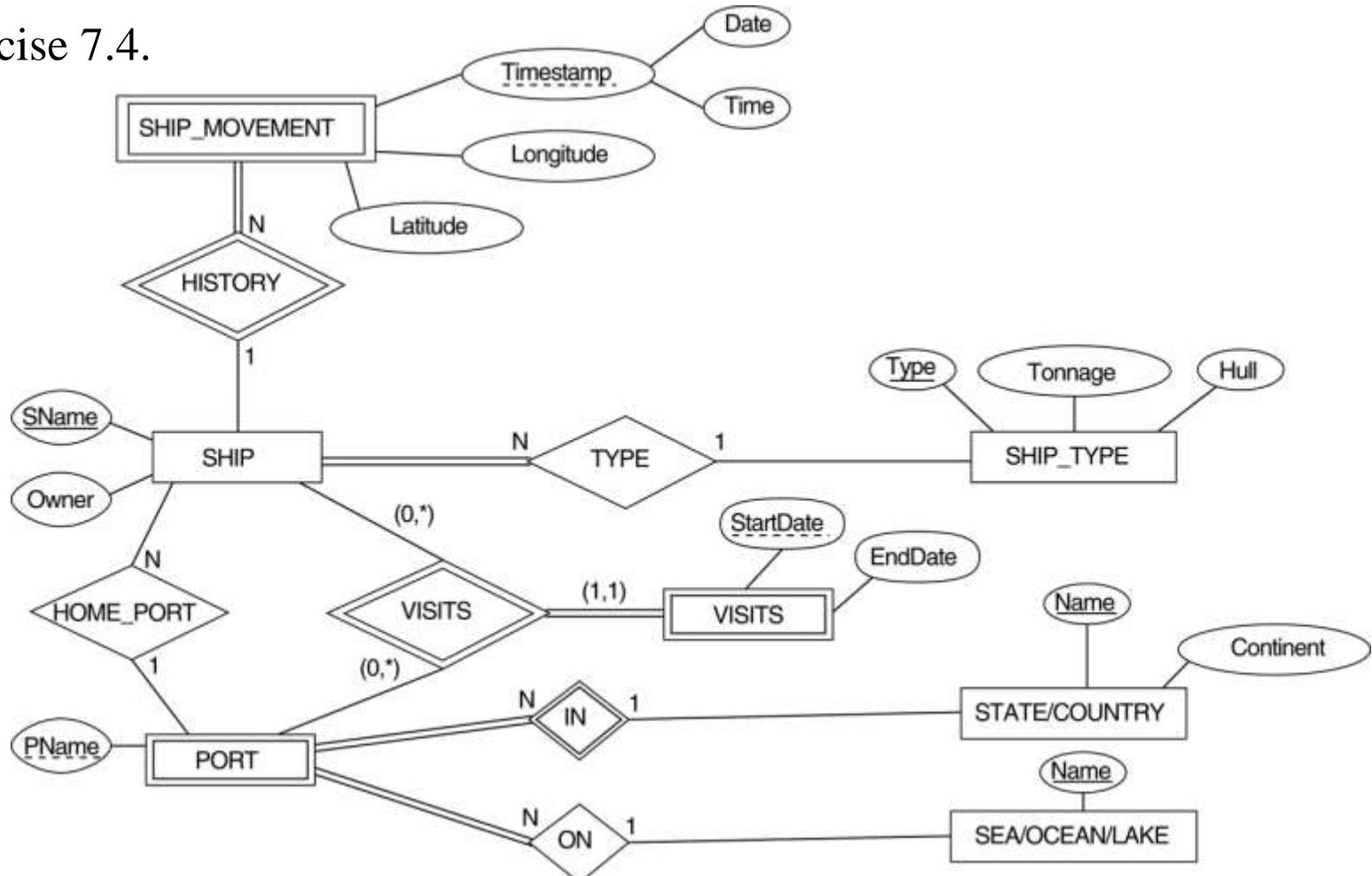


FIGURE 7.7

An ER schema for a SHIP_TRACKING database.

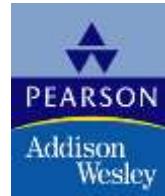
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 8

SQL-99: Schema Definition, Basic Constraints, and Queries



Data Definition, Constraints, and Schema Changes

- Used to CREATE, DROP, and ALTER the descriptions of the tables (relations) of a database

CREATE TABLE

- Specifies a new base relation by giving it a name, and specifying each of its attributes and their data types (INTEGER, FLOAT, DECIMAL(i,j), CHAR(n), VARCHAR(n))
- A constraint NOT NULL may be specified on an attribute

CREATE TABLE DEPARTMENT

```
(      DNAME      VARCHAR(10) NOT NULL,  
      DNUMBER     INTEGER      NOT NULL,  
      MGRSSN      CHAR(9),  
      MGRSTARTDATE  CHAR(9) );
```

CREATE TABLE

- In SQL2, can use the CREATE TABLE command for specifying the primary key attributes, secondary keys, and referential integrity constraints (foreign keys).
- Key attributes can be specified via the PRIMARY KEY and UNIQUE phrases

```
CREATE TABLE DEPT
(  DNAME      VARCHAR(10) NOT NULL,
   DNUMBER     INTEGER      NOT NULL,
   MGRSSN      CHAR(9),
   MGRSTARTDATE CHAR(9),
   PRIMARY KEY (DNUMBER),
   UNIQUE (DNAME),
   FOREIGN KEY (MGRSSN) REFERENCES EMP );
```

DROP TABLE

- Used to remove a relation (base table) *and its definition*
- The relation can no longer be used in queries, updates, or any other commands since its description no longer exists
- Example:

DROP TABLE DEPENDENT;

ALTER TABLE

- Used to add an attribute to one of the base relations
- The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is *not allowed* for such an attribute
- Example:

```
ALTER TABLE EMPLOYEE ADD JOB  
VARCHAR(12);
```

- The database users must still enter a value for the new attribute JOB for each EMPLOYEE tuple. This can be done using the UPDATE command

Features Added in SQL2 and SQL-99

- **CREATE SCHEMA**
- **REFERENTIAL INTEGRITY OPTIONS**

CREATE SCHEMA

- Specifies a new database schema by giving it a name

REFERENTIAL INTEGRITY OPTIONS

- We can specify RESTRICT, CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)

```
CREATE TABLE DEPT
(  DNAME      VARCHAR(10)      NOT NULL,
   DNUMBER    INTEGER      NOT NULL,
   MGRSSN     CHAR(9),
   MGRSTARTDATE CHAR(9),
   PRIMARY KEY (DNUMBER),
   UNIQUE (DNAME),
   FOREIGN KEY (MGRSSN) REFERENCES EMP
ON DELETE SET DEFAULT ON UPDATE
CASCADE );
```

REFERENTIAL INTEGRITY OPTIONS (continued)

```
CREATE TABLE EMP
(
    ENAME      VARCHAR(30) NOT NULL,
    ESSN      CHAR(9),
    BDATE DATE,
    DNO      INTEGER DEFAULT 1,
    SUPERSSN   CHAR(9),
    PRIMARY KEY (ESSN),
    FOREIGN KEY (DNO) REFERENCES DEPT
    ON DELETE SET DEFAULT ON UPDATE CASCADE,
    FOREIGN KEY (SUPERSSN) REFERENCES EMP
    ON DELETE SET NULL ON UPDATE CASCADE );
```

Additional Data Types in SQL2 and SQL-99

Has DATE, TIME, and TIMESTAMP data types

- **DATE:**

- Made up of year-month-day in the format yyyy-mm-dd

- **TIME:**

- Made up of hour:minute:second in the format hh:mm:ss

- **TIME(i):**

- Made up of hour:minute:second plus i additional digits specifying fractions of a second
 - format is hh:mm:ss:ii...i

- **TIMESTAMP:**

- Has both DATE and TIME components

Additional Data Types in SQL2 and SQL-99 (cont.)

● **INTERVAL:**

- Specifies a relative value rather than an absolute value
- Can be DAY/TIME intervals or YEAR/MONTH intervals
- Can be positive or negative when added to or subtracted from an absolute value, the result is an absolute value

Retrieval Queries in SQL

- SQL has one basic statement for retrieving information from a database; the SELECT statement
- This is *not the same as* the SELECT operation of the relational algebra
- Important distinction between SQL and the formal relational model; SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values
- Hence, an SQL relation (table) is a *multi-set* (sometimes called a bag) of tuples; it *is not* a set of tuples
- SQL relations can be constrained to be sets by specifying PRIMARY KEY or UNIQUE attributes, or by using the DISTINCT option in a query

Retrieval Queries in SQL (cont.)

- Basic form of the SQL SELECT statement is called a *mapping* or a *SELECT-FROM-WHERE block*

SELECT <attribute list>

FROM <table list>

WHERE <condition>

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

EMPLOYEE

FNAME	MINIT	LNAME	<u>SSN</u>	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
-------	-------	-------	------------	-------	---------	-----	--------	----------	-----

DEPARTMENT

DNAME	<u>DNUMBER</u>	MGRSSN	MGRSTARTDATE
-------	----------------	--------	--------------

DEPT_LOCATIONS

DNUMBER	DLOCATION
---------	-----------

PROJECT

PNAME	<u>PNUMBER</u>	PLOCATION	DNUM
-------	----------------	-----------	------

WORKS_ON

ESSN	PNO	HOURS
------	-----	-------

DEPENDENT

<u>ESSN</u>	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
-------------	----------------	-----	-------	--------------

Populated Database--Fig.5.6

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	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	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4	
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4	
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	MGRSSN	MGRSTARTDATE	DEPT_LOCATIONS	DNUMBER	DLOCATION
Research	5	333445555	1988-05-22		1	Houston	
Administration	4	987654321	1995-01-01		4	Stafford	
Headquarters	1	888665555	1981-06-19		5	Bellaire	
					5	Sugarland	
					5	Houston	

WORKS_ON	ESSN	PNO	HOURS
123456789	1	32.5	
123456789	2	7.5	
666884444	3	40.0	
453453453	1	20.0	
453453453	2	20.0	
333445555	2	10.0	
333445555	3	10.0	
333445555	10	10.0	
333445555	20	10.0	
999887777	30	30.0	
999887777	10	10.0	
987987987	10	35.0	
987987987	30	5.0	
987654321	30	20.0	
987654321	20	15.0	
888665555	20	null	

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5	
ProductY	2	Sugarland	5	
ProductZ	3	Houston	5	
Computerization	10	Stafford	4	
Reorganization	20	Houston	1	
Newbenefits	30	Stafford	4	

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
333445555		Alice	F	1986-04-05	DAUGHTER
333445555		Theodore	M	1983-10-25	SON
333445555		Joy	F	1958-05-03	SPOUSE
987654321		Abner	M	1942-02-28	SPOUSE
123456789		Michael	M	1988-01-04	SON
123456789		Alice	F	1988-12-30	DAUGHTER
123456789		Elizabeth	F	1967-05-05	SPOUSE

Simple SQL Queries

- Basic SQL queries correspond to using the SELECT, PROJECT, and JOIN operations of the relational algebra
- All subsequent examples use the COMPANY database
- Example of a simple query on *one* relation
- Query 0: Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.

Q0: **SELECT** **BDATE, ADDRESS**
 FROM **EMPLOYEE**
 WHERE FNAME='John' AND MINIT='B'
 AND **LNAME='Smith'**

- Similar to a SELECT-PROJECT pair of relational algebra operations; the SELECT-clause specifies the *projection attributes* and the WHERE-clause specifies the *selection condition*
- However, the result of the query *may contain* duplicate tuples

Simple SQL Queries (cont.)

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

```
Q1: SELECT      FNAME, LNAME, ADDRESS  
        FROM EMPLOYEE, DEPARTMENT  
       WHERE      DNAME='Research' AND  
      DNUMBER=DNO
```

- Similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations
- (*DNAME='Research'*) is a *selection condition* (corresponds to a SELECT operation in relational algebra)
- (*DNUMBER=DNO*) is a *join condition* (corresponds to a JOIN operation in relational algebra)

Simple SQL Queries (cont.)

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

**Q2: SELECT
FROM
WHERE
AND**

**PNUMBER, DNUM, LNAME, BDATE, ADDRESS
PROJECT, DEPARTMENT, EMPLOYEE
DNUM=DNUMBER AND MGRSSN=SSN
PLOCATION='Stafford'**

- In Q2, there are *two* join conditions
- The join condition DNUM=DNUMBER relates a project to its controlling department
- The join condition MGRSSN=SSN relates the controlling department to the employee who manages that department

Aliases, * and DISTINCT, Empty WHERE-clause

- In SQL, we can use the same name for two (or more) attributes as long as the attributes are in *different relations*.
A query that refers to two or more attributes with the same name must *qualify* the attribute name with the relation name by *prefixing* the relation name to the attribute name.

Example:

- EMPLOYEE.LNAME, DEPARTMENT.DNAME

ALIASES

- Some queries need to refer to the same relation twice
- In this case, *aliases* are given to the relation name
- Query 8: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

**Q8: SELECT
FROM
WHERE**

**E.FNAME, E.LNAME, S.FNAME,
S.LNAME
EMPLOYEE E S
E.SUPERSSN=S.SSN**

- In Q8, the alternate relation names E and S are called *aliases* or *tuple variables* for the EMPLOYEE relation
- We can think of E and S as two *different copies* of EMPLOYEE; E represents employees in role of *supervisees* and S represents employees in role of *supervisors*

ALIASES (cont.)

- Aliasing can also be used in any SQL query for convenience
Can also use the AS keyword to specify aliases

Q8: **SELECT** **E.FNAME, E.LNAME, S.FNAME,**
 S.LNAME
FROM **EMPLOYEE AS E, EMPLOYEE AS S**
WHERE **E.SUPERSSN=S.SSN**

UNSPECIFIED WHERE-clause

- A missing *WHERE-clause* indicates no condition; hence, *all tuples* of the relations in the *FROM-clause* are selected
- This is equivalent to the condition WHERE TRUE
- Query 9: Retrieve the SSN values for all employees.

Q9: **SELECT** **SSN**
 FROM **EMPLOYEE**

- If more than one relation is specified in the *FROM-clause* and there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected

UNSPECIFIED WHERE-clause (cont.)

- Example:

Q10: **SELECT** **SSN, DNAME**
 FROM **EMPLOYEE, DEPARTMENT**

- It is extremely important not to overlook specifying any selection and join conditions in the WHERE-clause; otherwise, incorrect and very large relations may result

USE OF *

- To retrieve all the attribute values of the selected tuples, a * is used, which stands for *all the attributes*

Examples:

Q1C: **SELECT** *
 FROM **EMPLOYEE**
 WHERE **DNO=5**

Q1D: **SELECT** *
 FROM **EMPLOYEE, DEPARTMENT**
 WHERE **DNAME='Research' AND**
 DNO=DNUMBER

USE OF DISTINCT

- SQL does not treat a relation as a set; *duplicate tuples can appear*
- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used
- For example, the result of Q11 may have duplicate SALARY values whereas Q11A does not have any duplicate values

Q11: **SELECT**
 FROM

SALARY
EMPLOYEE

Q11A: **SELECT**
 FROM

DISTINCT SALARY
EMPLOYEE

SET OPERATIONS

- SQL has directly incorporated some set operations
- There is a union operation (**UNION**), and in *some versions* of SQL there are set difference (**MINUS**) and intersection (**INTERSECT**) operations
- The resulting relations of these set operations are sets of tuples; *duplicate tuples are eliminated from the result*
- The set operations apply only to *union compatible relations*; the two relations must have the same attributes and the attributes must appear in the same order

SET OPERATIONS (cont.)

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

Q4: (SELECT PNAME

**FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUM=DNUMBER AND MGRSSN=SSN
AND LNAME='Smith')
UNION (SELECT PNAME
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE PNUMBER=PNO AND ESSN=SSN AND
LNAME='Smith')**

NESTING OF QUERIES

- A complete SELECT query, called a *nested query* , can be specified within the WHERE-clause of another query, called the *outer query*
- Many of the previous queries can be specified in an alternative form using nesting
- Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

**Q1: SELECT
 FROM
 WHERE
 FROM
 WHERE**

**FNAME, LNAME, ADDRESS
EMPLOYEE
DNO IN (SELECT DNUMBER
DEPARTMENT
DNAME='Research')**

NESTING OF QUERIES (cont.)

- The nested query selects the number of the 'Research' department
- The outer query select an EMPLOYEE tuple if its DNO value is in the result of either nested query
- The comparison operator **IN** compares a value v with a set (or multi-set) of values V, and evaluates to **TRUE** if v is one of the elements in V
- In general, we can have several levels of nested queries
- A reference to an *unqualified attribute* refers to the relation declared in the *innermost nested query*
- In this example, the nested query is *not correlated* with the outer query

CORRELATED NESTED QUERIES

- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query*, the two queries are said to be *correlated*
- The result of a correlated nested query is *different for each tuple (or combination of tuples) of the relation(s) the outer query*
- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

**Q12: SELECT
 FROM
 WHERE**

E.FNAME, E.LNAME
EMPLOYEE AS E
E.SSN IN (SELECT ESSN
 FROM DEPENDENT
 WHERE ESSN=E.SSN AND
 E.FNAME=DEPENDENT_NAME)

CORRELATED NESTED QUERIES (cont.)

- In Q12, the nested query has a different result *for each tuple* in the outer query
 - A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q12 may be written as in Q12A

Q12A: **SELECT**
FROM
WHERE **E.FNAME, E.LNAME**
EMPLOYEE E, DEPENDENT D
E.SSN=D.ESSN AND
E.FNAME=D.DEPENDENT_NAME

- The original SQL as specified for SYSTEM R also had a **CONTAINS** comparison operator, which is used in conjunction with nested correlated queries
 - This operator was dropped from the language, possibly because of the difficulty in implementing it efficiently

CORRELATED NESTED QUERIES (cont.)

- Most implementations of SQL *do not* have this operator
- The CONTAINS operator compares two *sets of values*, and returns TRUE if one set contains all values in the other set (reminiscent of the *division* operation of algebra).
 - Query 3: Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

Q3:

```
SELECT FNAME, LNAME
  FROM EMPLOYEE
 WHERE ( (SELECT          PNO
            FROM WORKS_ON
           WHERE          SSN=ESSN)
        CONTAINS
        (SELECT          PNUMBER
            FROM PROJECT
           WHERE          DNUM=5) )
```

CORRELATED NESTED QUERIES (cont.)

- In Q3, the second nested query, which is not correlated with the outer query, retrieves the project numbers of all projects controlled by department 5
- The first nested query, which is correlated, retrieves the project numbers on which the employee works, which is different *for each employee tuple* because of the correlation

THE EXISTS FUNCTION

- EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not
- We can formulate Query 12 in an alternative form that uses EXISTS as Q12B below

THE EXISTS FUNCTION (cont.)

- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

Q12B:

```
SELECT      FNAME, LNAME
FROM        EMPLOYEE
WHERE       EXISTS  (SELECT  *
                  FROM      DEPENDENT
                  WHERE     SSN=ESSN AND
                  FNAME=DEPENDENT_NAME)
```

THE EXISTS FUNCTION (cont.)

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

Q6:

```
SELECT          FNAME, LNAME
FROM           EMPLOYEE
WHERE          NOT EXISTS (SELECT *
                  FROM DEPENDENT
                  WHERE SSN=ESSN)
```

- In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected
- EXISTS is necessary for the expressive power of SQL

EXPLICIT SETS

- It is also possible to use an **explicit (enumerated)** set of **values** in the WHERE-clause rather than a nested query
- Query 13: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

Q13: **SELECT** **DISTINCT ESSN**
 FROM **WORKS_ON**
 WHERE **PNO IN (1, 2, 3)**

NULLS IN SQL QUERIES

- SQL allows queries that check if a value is NULL (missing or undefined or not applicable)
- SQL uses **IS** or **IS NOT** to compare NULLs because it considers each NULL value distinct from other NULL values, so equality comparison is not appropriate .
- Query 14: Retrieve the names of all employees who do not have supervisors.

Q14: **SELECT FNAME, LNAME**
 FROM EMPLOYEE
 WHERE SUPERSSN IS NULL

Note: If a join condition is specified, tuples with NULL values for the join attributes are not included in the result

Joined Relations Feature in SQL2

- Can specify a "joined relation" in the FROM-clause
- Looks like any other relation but is the result of a join
- Allows the user to specify different types of joins (regular "theta" JOIN, NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, CROSS JOIN, etc)

Joined Relations Feature in SQL2 (cont.)

- Examples:

Q8: **SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM EMPLOYEE E S
WHERE E.SUPERSSN=S.SSN**

can be written as:

Q8: **SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM (EMPLOYEE E LEFT OUTER JOIN EMPLOYEES
ON E.SUPERSSN=S.SSN)**

Q1: **SELECT FNAME, LNAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME='Research' AND DNUMBER=DNO**

Joined Relations Feature in SQL2 (cont.)

- could be written as:

```
Q1: SELECT      FNAME, LNAME, ADDRESS
      FROM (EMPLOYEE JOIN DEPARTMENT
             ON DNUMBER=DNO)
      WHERE      DNAME='Research'
```

or as:

```
Q1: SELECT      FNAME, LNAME, ADDRESS
      FROM (EMPLOYEE NATURAL JOIN DEPARTMENT
             AS DEPT(DNAME, DNO, MSSN, MSDATE))
      WHERE      DNAME='Research'
```

Joined Relations Feature in SQL2 (cont.)

- Another Example;
 - Q2 could be written as follows; this illustrates multiple joins in the joined tables

**Q2: SELECT PNUMBER, DNUM,
LNAME, BDATE,
ADDRESS
FROM
(PROJECT JOIN
DEPARTMENT ON
DNUM=DNUMBER) JOIN
EMPLOYEE ON
MGRSSN=SSN)
WHERE PLOCATION='Stafford'**

AGGREGATE FUNCTIONS

- Include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**
- Query 15: Find the maximum salary, the minimum salary, and the average salary among all employees.

Q15: **SELECT** **MAX(SALARY),**
 FROM **MIN(SALARY), AVG(SALARY)**
 EMPLOYEE

- Some SQL implementations *may not allow more than one function* in the SELECT-clause

AGGREGATE FUNCTIONS (cont.)

- Query 16: Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

**Q16: SELECT MAX(SALARY), MIN(SALARY),
 FROM AVG(SALARY)
 WHERE EMPLOYEE, DEPARTMENT
 DNO=DNUMBER AND
 DNAME='Research'**

AGGREGATE FUNCTIONS (cont.)

- Queries 17 and 18: Retrieve the total number of employees in the company (Q17), and the number of employees in the 'Research' department (Q18).

Q17: **SELECT** **COUNT (*)**
 FROM **EMPLOYEE**

Q18: **SELECT** **COUNT (*)**
 FROM **EMPLOYEE,**
 WHERE **DEPARTMENT**
 DNO=DNUMBER AND
 DNAME='Research'

GROUPING

- In many cases, we want to apply the aggregate functions *to subgroups of tuples in a relation*
- Each subgroup of tuples consists of the set of tuples that have *the same value* for the *grouping attribute(s)*
- The function is applied to each subgroup independently
- SQL has a **GROUP BY**-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*

GROUPING (cont.)

- Query 20: For each department, retrieve the department number, the number of employees in the department, and their average salary.

**Q20: SELECT DNO, COUNT (*), AVG (SALARY)
FROM EMPLOYEE
GROUP BY DNO**

- In Q20, the EMPLOYEE tuples are divided into groups--each group having the same value for the grouping attribute DNO
- The COUNT and AVG functions are applied to each such group of tuples separately
- The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples
- A join condition can be used in conjunction with grouping

GROUPING (cont.)

- Query 21: For each project, retrieve the project number, project name, and the number of employees who work on that project.

Q21:

```
SELECT      PNUMBER, PNAME, COUNT (*)
FROM        PROJECT, WORKS_ON
WHERE       PNUMBER=PNO
GROUP BY    PNUMBER, PNAME
```

- In this case, the grouping and functions are applied *after* the joining of the two relations

THE HAVING-CLAUSE

- Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*
- The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples)

THE HAVING-CLAUSE (cont.)

- Query 22: For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project.

Q22:

SELECT	PNUMBER, PNAME, COUNT (*)
FROM	PROJECT, WORKS_ON
WHERE	PNUMBER=PNO
GROUP BY	PNUMBER, PNAME
HAVING	COUNT (*) > 2

SUBSTRING COMPARISON

- The **LIKE** comparison operator is used to compare partial strings
- Two reserved characters are used: '%' (or '*' in some implementations) replaces an arbitrary number of characters, and '_' replaces a single arbitrary character

SUBSTRING COMPARISON (cont.)

- Query 25: Retrieve all employees whose address is in Houston, Texas. Here, the value of the ADDRESS attribute must contain the substring 'Houston, TX'.

Q25:

```
SELECT          FNAME, LNAME  
FROM           EMPLOYEE  
WHERE          ADDRESS LIKE  
                  '%Houston,TX%'
```

SUBSTRING COMPARISON (cont.)

- Query 26: Retrieve all employees who were born during the 1950s. Here, '5' must be the 8th character of the string (according to our format for date), so the BDATE value is ' 5 ', with each underscore as a place holder for a single arbitrary character.

Q26: **SELECT** **FNAME, LNAME**
 FROM **EMPLOYEE**
 WHERE **BDATE LIKE** **' 5 '**

- The LIKE operator allows us to get around the fact that each value is considered atomic and indivisible; hence, in SQL, character string attribute values are not atomic

ARITHMETIC OPERATIONS

- The standard arithmetic operators '+', '-'. '*', and '/' (for addition, subtraction, multiplication, and division, respectively) can be applied to numeric values in an SQL query result
- Query 27: Show the effect of giving all employees who work on the 'ProductX' project a 10% raise.

Q27: SELECT

WHERE

**FNAME, LNAME, 1.1*SALARY
FROM EMPLOYEE, WORKS_ON, PROJECT
SSN=ESSN AND PNO=PNUMBER AND
PNAME='ProductX'**

ORDER BY

- The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s)
- Query 28: Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name.

Q28: **SELECT**
 FROM

 WHERE
 AND
 ORDER BY **DNAME, LNAME, FNAME, PNAME**
 DEPARTMENT, EMPLOYEE,
 WORKS_ON, PROJECT
 DNUMBER=DNO AND SSN=ESSN
 PNO=PNUMBER
 DNAME, LNAME

ORDER BY (cont.)

- The default order is in ascending order of values
- We can specify the keyword **DESC** if we want a descending order; the keyword **ASC** can be used to explicitly specify ascending order, even though it is the default

Summary of SQL Queries

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

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

Summary of SQL Queries (cont.)

- The SELECT-clause lists the attributes or functions to be retrieved
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- GROUP BY specifies grouping attributes
- HAVING specifies a condition for selection of groups
- ORDER BY specifies an order for displaying the result of a query
- A query is evaluated by first applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause

Specifying Updates in SQL

- There are three SQL commands to modify the database; INSERT, DELETE, and UPDATE

INSERT

- In its simplest form, it is used to add one or more tuples to a relation
- Attribute values should be listed in the same order as the attributes were specified in the CREATE TABLE command

INSERT (cont.)

- Example:

U1: INSERT INTO EMPLOYEE

**VALUES ('Richard','K','Marini', '653298653', '30-DEC-52',
'98 Oak Forest,Katy,TX', 'M', 37000,'987654321', 4)**

- An alternate form of INSERT specifies explicitly the attribute names that correspond to the values in the new tuple
- Attributes with NULL values can be left out
- Example: Insert a tuple for a new EMPLOYEE for whom we only know the FNAME, LNAME, and SSN attributes.

U1A: INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)

VALUES ('Richard', 'Marini', '653298653')

INSERT (cont.)

- Important Note: Only the constraints specified in the DDL commands are automatically enforced by the DBMS when updates are applied to the database
- Another variation of INSERT allows insertion of *multiple tuples* resulting from a query into a relation

INSERT (cont.)

- Example: Suppose we want to create a temporary table that has the name, number of employees, and total salaries for each department. A table DEPTS_INFO is created by U3A, and is loaded with the summary information retrieved from the database by the query in U3B.

U3A: **CREATE TABLE DEPTS_INFO**
 (**DEPT_NAME** **VARCHAR(10)**,
 NO_OF_EMPS **INTEGER**,
 TOTAL_SAL **INTEGER**);

U3B: **INSERT INTO DEPTS_INFO (DEPT_NAME,**
 NO_OF_EMPS, TOTAL_SAL)
 SELECT DNAME, COUNT (*), SUM (SALARY)
 FROM DEPARTMENT, EMPLOYEE
 WHERE DNUMBER=DNO
 GROUP BY DNAME ;

INSERT (cont.)

- Note: The DEPTS_INFO table may not be up-to-date if we change the tuples in either the DEPARTMENT or the EMPLOYEE relations *after* issuing U3B. We have to create a view (see later) to keep such a table up to date.

DELETE

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

DELETE (cont.)

- Examples:

U4A: **DELETE FROM** **EMPLOYEE**
WHERE **LNAME='Brown'**

U4B: **DELETE FROM** **EMPLOYEE**
WHERE **SSN='123456789'**

U4C: **DELETE FROM** **EMPLOYEE**
WHERE **DNO IN**
 (SELECT **DNUMBER**
 FROM **DEPARTMENT**
 WHERE **DNAME='Research')**

U4D: **DELETE FROM** **EMPLOYEE**

Elmasri et al., *Database Systems*, Fourth Edition

Copyright © 2004 Pearson Education, Inc.

UPDATE

- Used to modify attribute values of one or more selected tuples
- A WHERE-clause selects the tuples to be modified
- An additional SET-clause specifies the attributes to be modified and their new values
- Each command modifies tuples *in the same relation*
- Referential integrity should be enforced

UPDATE (cont.)

- Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

**U5: UPDATE
SET
WHERE**

**PROJECT
PLOCATION = 'Bellaire', DNUM = 5
PNUMBER=10**

UPDATE (cont.)

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

**U6: UPDATE
SET
WHERE**

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

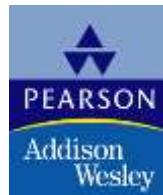
- In this request, the modified SALARY value depends on the original SALARY value in each tuple
- The reference to the SALARY attribute on the right of = refers to the old SALARY value before modification
- The reference to the SALARY attribute on the left of = refers to the new SALARY value after modification

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

MORE SQL: Assertions, Views, and Programming Techniques



Chapter Outline

- 9.1 General Constraints as Assertions**
- 9.2 Views in SQL**
- 9.3 Database Programming**
- 9.4 Embedded SQL**
- 9.5 Functions Calls, SQL/CLI**
- 9.6 Stored Procedures, SQL/PSM**
- 9.7 Summary**

Chapter Objectives

- Specification of more general constraints via assertions
- SQL facilities for defining views (virtual tables)
- Various techniques for accessing and manipulating a database via programs in general-purpose languages (e.g., Java)

Constraints as Assertions

- General constraints: constraints that do not fit in the basic SQL categories (presented in chapter 8)
- Mechanism: `CREATE ASSERTION`
 - components include: a constraint name, followed by `CHECK`, followed by a condition

Assertions: An Example

- “The salary of an employee must not be greater than the salary of the manager of the department that the employee works for”

```
CREATE ASSERTION SALARY_CONSTRAINT  
CHECK (NOT EXISTS (SELECT *  
FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT  
D  
WHERE E.SALARY > M.SALARY AND  
E.DNO=D.NUMBER AND  
D.MGRSSN=M.SSN))
```

Using General Assertions

- Specify a query that violates the condition; include inside a NOT EXISTS clause
- Query result must be empty
 - if the query result is not empty, the assertion has been violated

SQL Triggers

- Objective: to monitor a database and take action when a condition occurs
- Triggers are expressed in a syntax similar to assertions and include the following:
 - event (e.g., an update operation)
 - condition
 - action (to be taken when the condition is satisfied)

SQL Triggers: An Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

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

Views in SQL

- A view is a “virtual” table that is derived from other tables
- Allows for limited update operations (since the table may not physically be stored)
- Allows full query operations
- A convenience for expressing certain operations

Specification of Views

- SQL command: CREATE VIEW
 - a table (view) name
 - a possible list of attribute names (for example, when arithmetic operations are specified or when we want the names to be different from the attributes in the base relations)
 - a query to specify the table contents

SQL Views: An Example

- Specify a different WORKS_ON table

```
CREATE TABLE WORKS_ON_NEW AS  
SELECT FNAME, LNAME, PNAME, HOURS  
      FROM EMPLOYEE, PROJECT, WORKS_ON  
     WHERE SSN=ESSN AND PNO=PNUMBER  
   GROUP BY PNAME;
```

Using a Virtual Table

- We can specify SQL queries on a newly create table (view):

```
SELECT FNAME, LNAME FROM WORKS_ON_NEW  
WHERE PNAME='Seena' ;
```

- When no longer needed, a view can be dropped:

```
DROP WORKS_ON_NEW;
```

Efficient View Implementation

- Query modification: present the view query in terms of a query on the underlying base tables
 - disadvantage: inefficient for views defined via complex queries (especially if additional queries are to be applied to the view within a short time period)

Efficient View Implementation

- View materialization: involves physically creating and keeping a temporary table
 - assumption: other queries on the view will follow
 - concerns: maintaining correspondence between the base table and the view when the base table is updated
 - strategy: incremental update

View Update

- Update on a single view without aggregate operations: update may map to an update on the underlying base table
- Views involving joins: an update *may* map to an update on the underlying base relations
 - not always possible

Un-updatable Views

- Views defined using groups and aggregate functions are not updateable
- Views defined on multiple tables using joins are generally not updateable
- WITH CHECK OPTION: must be added to the definition of a view if the view is to be updated
 - to allow check for updatability and to plan for an execution strategy

Database Programming

- Objective: to access a database from an application program (as opposed to interactive interfaces)
- Why? An interactive interface is convenient but not sufficient; a majority of database operations are made thru application programs (nowadays thru web applications)

Database Programming Approaches

- Embedded commands: database commands are embedded in a general-purpose programming language
- Library of database functions: available to the host language for database calls; known as an *API*
- A brand new, full-fledged language (minimizes impedance mismatch)

Impedance Mismatch

- Incompatibilities between a host programming language and the database model, e.g.,
 - type mismatch and incompatibilities; requires a new binding for each language
 - set vs. record-at-a-time processing
 - need special iterators to loop over query results and manipulate individual values

Steps in Database Programming

1. Client program opens a connection to the database server
2. Client program submits queries to and/or updates the database
3. When database access is no longer needed, client program terminates the connection

Embedded SQL

- Most SQL statements can be embedded in a general-purpose *host* programming language such as COBOL, C, Java
- An embedded SQL statement is distinguished from the host language statements by EXEC SQL and a matching END-EXEC (or semicolon)
 - *shared variables* (used in both languages) usually prefixed with a colon (:) in SQL

Example: Variable Declaration in Language C

- Variables inside DECLARE are shared and can appear (while prefixed by a colon) in SQL statements
- SQLCODE is used to communicate errors/exceptions between the database and the program

```
int loop;  
  
EXEC SQL BEGIN DECLARE SECTION;  
varchar dname[16], fname[16], ...;  
char ssn[10], bdate[11], ...;  
int dno, dnumber, SQLCODE, ...;  
  
EXEC SQL END DECLARE SECTION;
```

SQL Commands for Connecting to a Database

- Connection (multiple connections are possible but only one is active)

```
CONNECT TO server-name AS connection-name  
AUTHORIZATION user-account-info;
```

- Change from an active connection to another one

```
SET CONNECTION connection-name;
```

- Disconnection

```
DISCONNECT connection-name;
```

Embedded SQL in C Programming Examples

```
loop = 1;
while (loop) {
    prompt ("Enter SSN: ", ssn);
    EXEC SQL
        select FNAME, LNAME, ADDRESS, SALARY
        into :fname, :lname, :address, :salary
        from EMPLOYEE where SSN == :ssn;
    if (SQLCODE == 0) printf(fname, ...);
    else printf("SSN does not exist: ", ssn);
    prompt("More SSN? (1=yes, 0=no) : ", loop);
END-EXEC
}
```

Embedded SQL in C Programming Examples

- A *cursor* (iterator) is needed to process multiple tuples
- FETCH commands move the cursor to the next tuple
- CLOSE CURSOR indicates that the processing of query results has been completed

Dynamic SQL

- Objective: executing new (not previously compiled) SQL statements at run-time
 - a program accepts SQL statements from the keyboard at run-time
 - a point-and-click operation translates to certain SQL query
- Dynamic update is relatively simple; dynamic query can be complex
 - because the type and number of retrieved attributes are unknown at compile time

Dynamic SQL: An Example

```
EXEC SQL BEGIN DECLARE SECTION;
varchar sqlupdatestring[256];
EXEC SQL END DECLARE SECTION;
...
prompt ("Enter update command:",
       sqlupdatestring);
EXEC SQL PREPARE sqlcommand FROM
       :sqlupdatestring;
EXEC SQL EXECUTE sqlcommand;
```

Embedded SQL in Java

- SQLJ: a standard for embedding SQL in Java
- An SQLJ translator converts SQL statements into Java (to be executed thru the *JDBC* interface)
- Certain classes, e.g., `java.sql` have to be imported

Java Database Connectivity

- JDBC: SQL connection function calls for Java programming
- A Java program with JDBC functions can access any relational DBMS that has a JDBC driver
- JDBC allows a program to connect to several databases (known as *data sources*)

Steps in JDBC Database Access

1. Import JDBC library (`java.sql.*`)
2. Load JDBC driver:
`Class.forName("oracle.jdbc.driver.OracleDriver")`
3. Define appropriate variables
4. Create a connect object (via `getConnection`)
5. Create a statement object from the Statement class:
 1. `PreparedStatement`
 2. `CallableStatement`

Steps in JDBC Database Access (continued)

6. Identify statement parameters (to be designated by question marks)
7. Bound parameters to program variables
8. Execute SQL statement (referenced by an object) via JDBC's `executeQuery`
9. Process query results (returned in an object of type `ResultSet`)
 - `ResultSet` is a 2-dimentional table

Embedded SQL in Java: An Example

```
ssn = readEntry("Enter a SSN: ");
try {
    #sql{select FNAME< LNAME, ADDRESS, SALARY
    into :fname, :lname, :address, :salary
    from EMPLOYEE where SSN = :ssn};
}
catch (SQLException se) {
    System.out.println("SSN does not exist:
    ",+ssn);
    return;
}
System.out.println(fname+" "+lname+... );
```

Multiple Tuples in SQLJ

- SQLJ supports two types of iterators:
 - *named iterator*: associated with a query result
 - *positional iterator*: lists only attribute types in a query result
- A `FETCH` operation retrieves the next tuple in a query result:

```
fetch iterator-variable into program-variable
```

Database Programming with Functional Calls

- Embedded SQL provides static database programming
- API: dynamic database programming with a library of functions
 - advantage: no preprocessor needed (thus more flexible)
 - drawback: SQL syntax checks to be done at run-time

SQL Call Level Interface

- A part of the SQL standard
- Provides easy access to several databases within the same program
- Certain libraries (e.g., `sqlcli.h` for C) have to be installed and available
- SQL statements are dynamically created and passed as string parameters in the calls

Components of SQL/CLI

- *Environment record*: keeps track of database connections
- *Connection record*: keep tracks of info needed for a particular connection
- *Statement record*: keeps track of info needed for one SQL statement
- *Description record*: keeps track of tuples

Steps in C and SQL/CLI Programming

1. Load SQL/CLI libraries
2. Declare record handle variables for the above components (called: SQLHSTMT, SQLHDBC, SQLHENV, SQLHDEC)
3. Set up an environment record using SQLAllocHandle
4. Set up a connection record using SQLAllocHandle
5. Set up a statement record using SQLAllocHandle

Steps in C and SQL/CLI Programming (continued)

6. Prepare a statement using SQL/CLI function `SQLPrepare`
7. Bound parameters to program variables
8. Execute SQL statement via `SQLExecute`
9. Bound columns in a query to a C variable via `SQLBindCol`
10. Use `SQLFetch` to retrieve column values into C variables

Database Stored Procedures

- Persistent procedures/functions (modules) are stored locally and executed by the database server (as opposed to execution by clients)
- Advantages:
 - if the procedure is needed by many applications, it can be invoked by any of them (thus reduce duplications)
 - execution by the server reduces communication costs
 - enhance the modeling power of views

Stored Procedure Constructs

- A stored procedure

```
CREATE PROCEDURE procedure-name (params)
local-declarations
procedure-body;
```

- A stored function

```
CREATE FUNCTION fun-name (params) RETURNS return-type
local-declarations
function-body;
```

- Calling a procedure or function

```
CALL procedure-name/fun-name (arguments);
```

SQL Persistent Stored Modules

- SQL/PSM: part of the SQL standard for writing persistent stored modules
- SQL + stored procedures/functions + additional programming constructs
 - e.g., branching and looping statements
 - enhance the power of SQL

SQL/PSM: An Example

```
CREATE FUNCTION DEPT_SIZE (IN deptno INTEGER)
RETURNS VARCHAR[7]
DECLARE TOT_EMPS INTEGER;

SELECT COUNT (*) INTO TOT_EMPS
FROM SELECT EMPLOYEE WHERE DNO = deptno;
IF TOT_EMPS > 100 THEN RETURN "HUGE"
ELSEIF TOT_EMPS > 50 THEN RETURN "LARGE"
ELSEIF TOT_EMPS > 30 THEN RETURN "MEDIUM"
ELSE RETURN "SMALL"
ENDIF;
```

Summary

- Assertions provide a means to specify additional constraints
- Triggers are a special kind of assertions; they define actions to be taken when certain conditions occur
- Views are a convenient means for creating temporary (virtual) tables

Summary (continued)

- A database may be accessed via an interactive database
- Most often, however, data in a database is manipulate via application programs
- Several methods of database programming:
 - embedded SQL
 - dynamic SQL
 - stored procedure and function

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 10

Functional Dependencies and Normalization for Relational Databases



Chapter Outline

1 Informal Design Guidelines for Relational Databases

1.1 Semantics of the Relation Attributes

1.2 Redundant Information in Tuples and Update Anomalies

1.3 Null Values in Tuples

1.4 Spurious Tuples

2 Functional Dependencies (FDs)

2.1 Definition of FD

2.2 Inference Rules for FDs

2.3 Equivalence of Sets of FDs

2.4 Minimal Sets of FDs

Chapter Outline(contd.)

3 Normal Forms Based on Primary Keys

- 3.1 Normalization of Relations
- 3.2 Practical Use of Normal Forms
- 3.3 Definitions of Keys and Attributes Participating in Keys
- 3.4 First Normal Form
- 3.5 Second Normal Form
- 3.6 Third Normal Form

4 General Normal Form Definitions (For Multiple Keys)

5 BCNF (Boyce-Codd Normal Form)

1 Informal Design Guidelines for Relational Databases (1)

- What is relational database design?
 - The grouping of attributes to form "good" relation schemas
- Two levels of relation schemas
 - The logical "user view" level
 - The storage "base relation" level
- Design is concerned mainly with base relations
- What are the criteria for "good" base relations?

Informal Design Guidelines for Relational Databases (2)

- We first discuss informal guidelines for good relational design
- Then we discuss formal concepts of functional dependencies and normal forms
 - 1NF (First Normal Form)
 - 2NF (Second Normal Form)
 - 3NF (Third Normal Form)
 - BCNF (Boyce-Codd Normal Form)
- Additional types of dependencies, further normal forms, relational design algorithms by synthesis are discussed in Chapter 11

1.1 Semantics of the Relation Attributes

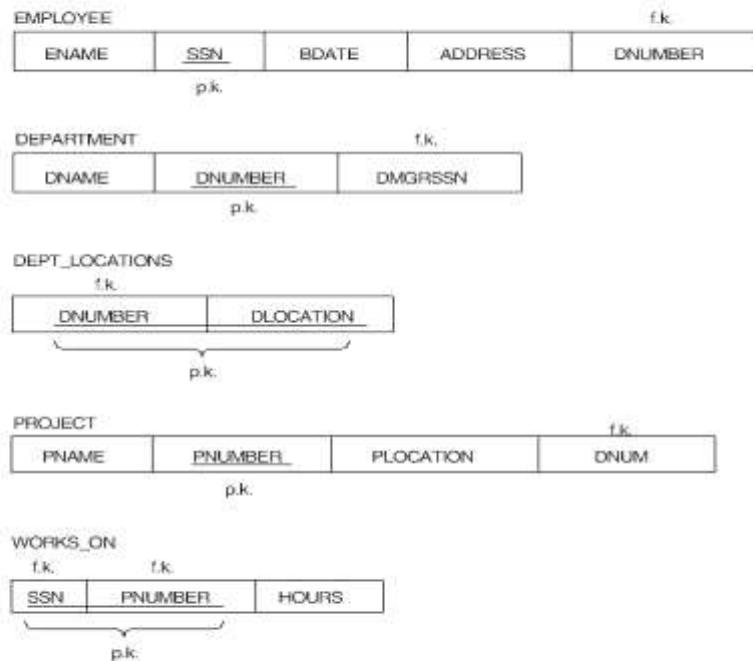
GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).

- Attributes of different entities (EMPLOYEES, DEPARTMENTS, PROJECTS) should not be mixed in the same relation
- Only foreign keys should be used to refer to other entities
- Entity and relationship attributes should be kept apart as much as possible.

Bottom Line: Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.

Figure 10.1 A simplified COMPANY relational database schema

Figure 14.1 Simplified version of the COMPANY relational database schema.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.1 in Edition 4

1.2 Redundant Information in Tuples and Update Anomalies

- Mixing attributes of multiple entities may cause problems
- Information is stored redundantly wasting storage
- Problems with update anomalies
 - Insertion anomalies
 - Deletion anomalies
 - Modification anomalies

EXAMPLE OF AN UPDATE ANOMALY (1)

Consider the relation:

EMP_PROJ (Emp#, Proj#, Ename, Pname, No_hours)

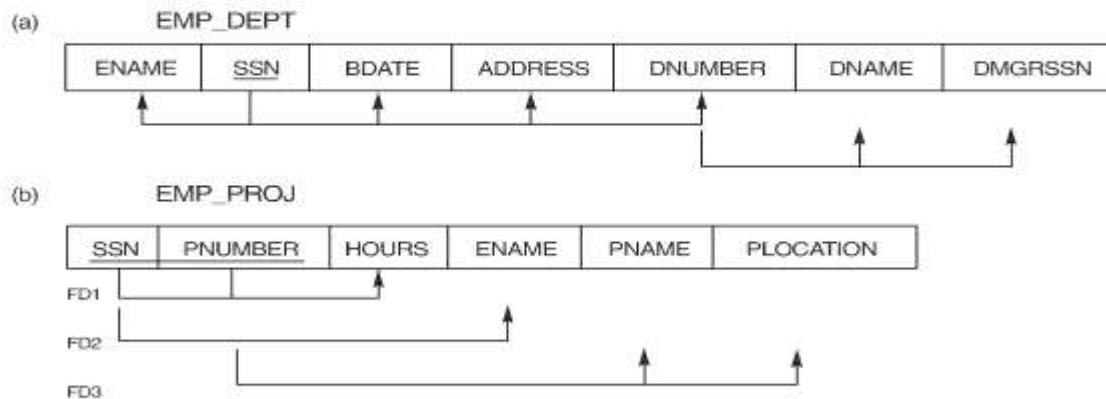
- **Update Anomaly:** Changing the name of project number P1 from “Billing” to “Customer-Accounting” may cause this update to be made for all 100 employees working on project P1.

EXAMPLE OF AN UPDATE ANOMALY (2)

- **Insert Anomaly:** Cannot insert a project unless an employee is assigned to .
Inversely - Cannot insert an employee unless an he/she is assigned to a project.
- **Delete Anomaly:** When a project is deleted, it will result in deleting all the employees who work on that project. Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

Figure 10.3 Two relation schemas suffering from update anomalies

Figure 14.3 Two relation schemas and their functional dependencies. Both suffer from update anomalies. (a) The EMP_DEPT relation schema. (b) The EMP_PROJ relation schema.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.3 in Edition 4

Figure 10.4 Example States for EMP_DEPT and EMP_PROJ

Figure 14.4 Example relations for the schemas in Figure 14.3 that result from applying NATURAL JOIN to the relations in Figure 14.2. These may be stored as base relations for performance reasons.

EMP_DEPT						
ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
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

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.4 in Edition 4

Guideline to Redundant Information in Tuples and Update Anomalies

- **GUIDELINE 2:** Design a schema that does not suffer from the insertion, deletion and update anomalies. If there are any present, then note them so that applications can be made to take them into account

1.3 Null Values in Tuples

GUIDELINE 3: Relations should be designed such that their tuples will have as few NULL values as possible

- Attributes that are NULL frequently could be placed in separate relations (with the primary key)
- Reasons for nulls:
 - attribute not applicable or invalid
 - attribute value unknown (may exist)
 - value known to exist, but unavailable

1.4 Spurious Tuples

- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The "lossless join" property is used to guarantee meaningful results for join operations

GUIDELINE 4: The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.

Spurious Tuples (2)

There are two important properties of decompositions:

- (a) non-additive or losslessness of the corresponding join
- (b) preservation of the functional dependencies.

Note that property (a) is extremely important and *cannot* be sacrificed. Property (b) is less stringent and may be sacrificed. (See Chapter 11).

2.1 Functional Dependencies (1)

- Functional dependencies (FDs) are used to specify *formal measures* of the "goodness" of relational designs
- FDs and keys are used to define **normal forms** for relations
- FDs are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y

Functional Dependencies (2)

- $X \rightarrow Y$ holds if whenever two tuples have the same value for X , they *must have* the same value for Y
- For any two tuples t_1 and t_2 in any relation instance $r(R)$:
If $t_1[X]=t_2[X]$, *then* $t_1[Y]=t_2[Y]$
- $X \rightarrow Y$ in R specifies a *constraint* on all relation instances $r(R)$
- Written as $X \rightarrow Y$; can be displayed graphically on a relation schema as in Figures. (denoted by the arrow:).
- FDs are derived from the real-world constraints on the attributes

Examples of FD constraints (1)

- social security number determines employee name
 $\text{SSN} \rightarrow \text{ENAME}$
- project number determines project name and location
 $\text{PNUMBER} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}$
- employee ssn and project number determines the hours per week that the employee works on the project
 $\{\text{SSN}, \text{PNUMBER}\} \rightarrow \text{HOURS}$

Examples of FD constraints (2)

- An FD is a property of the attributes in the schema R
- The constraint must hold on *every relation instance* $r(R)$
- If K is a key of R , then K functionally determines all attributes in R (since we never have two distinct tuples with $t1[K]=t2[K]$)

2.2 Inference Rules for FDs (1)

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

Armstrong's inference rules:

- IR1. (**Reflexive**) If $Y \subsetneq X$, then $X \rightarrow Y$
- IR2. (**Augmentation**) If $X \rightarrow Y$, then $XZ \rightarrow YZ$
(Notation: XZ stands for $X \cup Z$)
- IR3. (**Transitive**) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- IR1, IR2, IR3 form a *sound* and *complete* set of inference rules

Inference Rules for FDs (2)

Some additional inference rules that are useful:

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

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

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

- The last three inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property)

Inference Rules for FDs (3)

- **Closure** of a set F of FDs is the set F^+ of all FDs that can be inferred from F
- **Closure** of a set of attributes X with respect to F is the set X^+ of all attributes that are functionally determined by X
- X^+ can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F

2.3 Equivalence of Sets of FDs

- Two sets of FDs F and G are **equivalent** if:
 - every FD in F can be inferred from G, *and*
 - every FD in G can be inferred from F
- Hence, F and G are equivalent if $F^+ = G^+$
- Definition: F **covers** G if every FD in G can be inferred from F (i.e., if $G^+ \subseteq F^+$)
- F and G are equivalent if F covers G and G covers F
- There is an algorithm for checking equivalence of sets of FDs

2.4 Minimal Sets of FDs (1)

- A set of FDs is **minimal** if it satisfies the following conditions:
 - (1) Every dependency in F has a single attribute for its RHS.
 - (2) We cannot remove any dependency from F and have a set of dependencies that is equivalent to F.
 - (3) We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where Y proper-subset-of X (Y subset-of X) and still have a set of dependencies that is equivalent to F.

Minimal Sets of FDs (2)

- Every set of FDs has an equivalent minimal set
- There can be several equivalent minimal sets
- There is no simple algorithm for computing a minimal set of FDs that is equivalent to a set F of FDs
- To synthesize a set of relations, we assume that we start with a set of dependencies that is a minimal set (e.g., see algorithms 11.2 and 11.4)

3 Normal Forms Based on Primary Keys

- 3.1 Normalization of Relations
- 3.2 Practical Use of Normal Forms
- 3.3 Definitions of Keys and Attributes
 - Participating in Keys
- 3.4 First Normal Form
- 3.5 Second Normal Form
- 3.6 Third Normal Form

3.1 Normalization of Relations (1)

- **Normalization:** The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations
- **Normal form:** Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form

Normalization of Relations (2)

- 2NF, 3NF, BCNF based on keys and FDs of a relation schema
- 4NF based on keys, multi-valued dependencies : MVDs; 5NF based on keys, join dependencies : JDs (Chapter 11)
- Additional properties may be needed to ensure a good relational design (lossless join, dependency preservation; Chapter 11)

3.2 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
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are **hard to understand** or to **detect**
- The database designers ***need not*** normalize to the highest possible normal form. (usually up to 3NF, BCNF or 4NF)
- **Denormalization:** the process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

3.3 Definitions of Keys and Attributes Participating in Keys (1)

- A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes S subset-of R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
- A **key** K is a superkey with the *additional property* that removal of any attribute from K will cause K not to be a superkey any more.

Definitions of Keys and Attributes Participating in Keys (2)

- 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*.
- A **Prime attribute** must be a member of *some candidate key*
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

3.2 First Normal Form

- Disallows composite attributes, multivalued attributes, and **nested relations**; attributes whose values *for an individual tuple* are non-atomic
- Considered to be part of the definition of relation

Figure 10.8 Normalization into 1NF

Figure 14.8 Normalization into 1NF. (a) Relation schema that is not in 1NF. (b) Example relation instance. (c) 1NF relation with redundancy.

(a)

DEPARTMENT			
DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	[Bellaire, Sugarland, Houston]
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	[Houston]

(b)

DEPARTMENT			
DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	[Bellaire, Sugarland, Houston]
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	[Houston]

(c)

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.8 in Edition 4

Figure 10.9 Normalization nested relations into 1NF

Figure 14.9 Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a “nested relation” PROJS. (b) Example extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposing EMP_PROJ into 1NF relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

(a) EMP_PROJ

SSN	ENAME	PROJS	
		PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
656884444	Nareyan,Ramseh 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	Zelaja,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
8886665555	Borg,James E.	20	null

(b) EMP_PROJ

SSN	ENAME	PROJS	
		PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
656884444	Nareyan,Ramseh 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	Zelaja,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
8886665555	Borg,James E.	20	null

(c)

EMP_PROJ1	EMP_PROJ2
SSN	ENAME

© Addison Wesley Longman, Inc. 2000. Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.9 in Edition 4

3.3 Second Normal Form (1)

- Uses the concepts of FDs, **primary key**

Definitions:

- **Prime attribute** - attribute that is member of the primary key K
- **Full functional dependency** - a FD $Y \rightarrow Z$ where removal of any attribute from Y means the FD does not hold any more

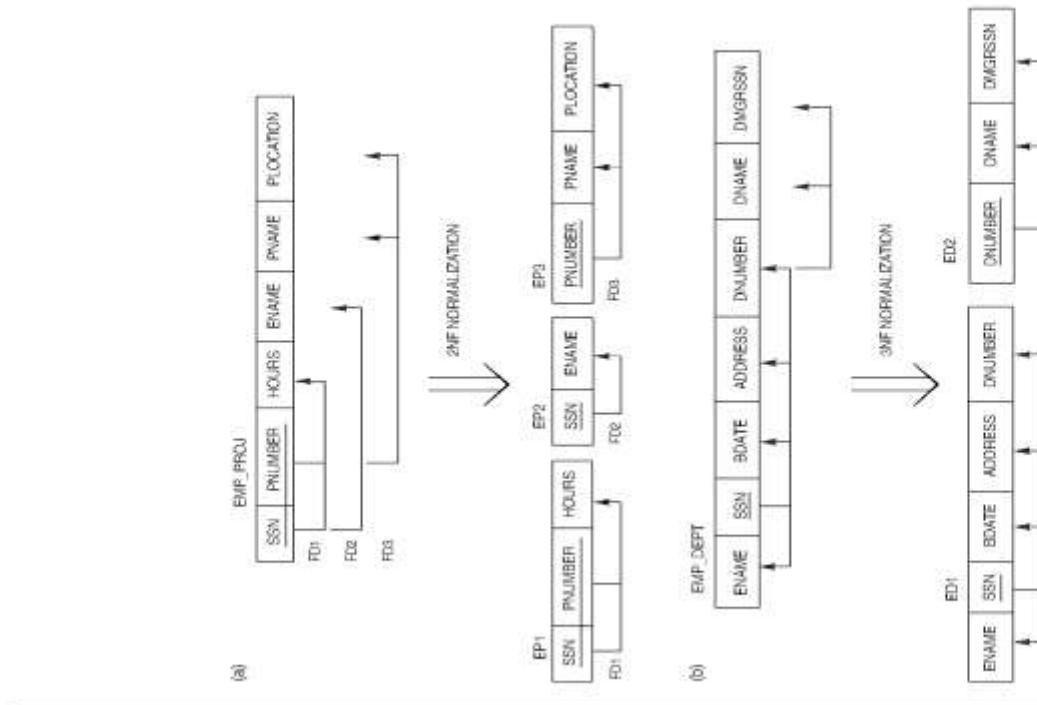
Examples: - $\{\text{SSN}, \text{PNUMBER}\} \rightarrow \text{HOURS}$ is a full FD since neither $\text{SSN} \rightarrow \text{HOURS}$ nor $\text{PNUMBER} \rightarrow \text{HOURS}$ hold
- $\{\text{SSN}, \text{PNUMBER}\} \rightarrow \text{ENAME}$ is *not* a full FD (it is called a *partial dependency*) since $\text{SSN} \rightarrow \text{ENAME}$ also holds

Second Normal Form (2)

- A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on the primary key
- R can be decomposed into 2NF relations via the process of 2NF normalization

Figure 10.10 Normalizing into 2NF and 3NF

Figure 14.10 The normalization process. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

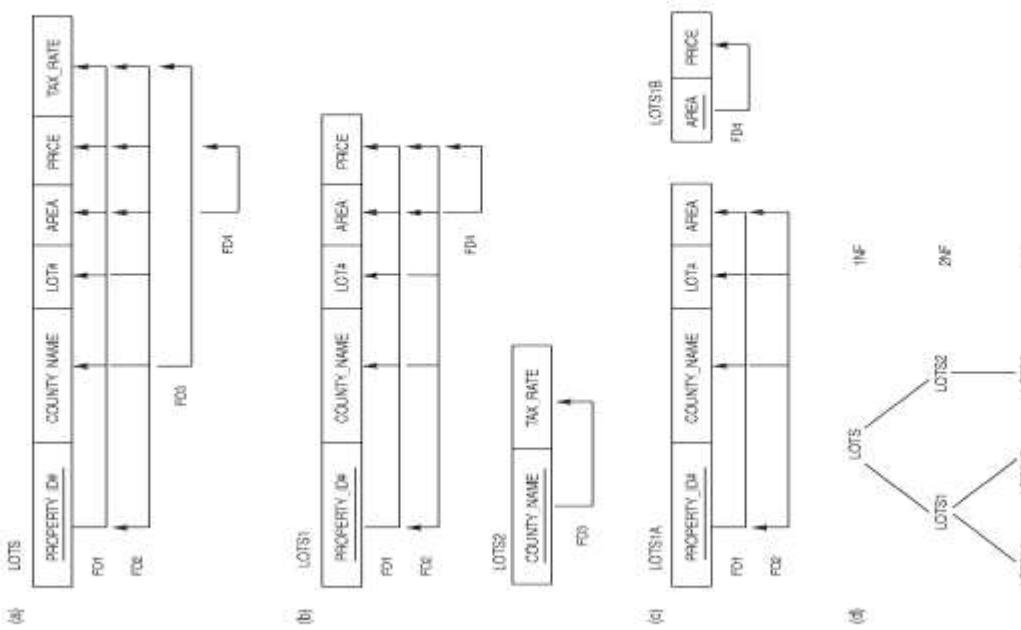


© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.10 in Edition 4

Figure 10.11 Normalization into 2NF and 3NF

Figure 14.11 Normalization to 2NF and 3NF. (a) The lots relation schema and its functional dependencies fd_1 through fd_4 . (b) Decomposing lots into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of normalization of lots.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.11 in Edition 4

3.4 Third Normal Form (1)

Definition:

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

Examples:

- SSN \rightarrow DMGRSSN is a *transitive* FD since SSN \rightarrow DNUMBER and DNUMBER \rightarrow DMGRSSN hold
- SSN \rightarrow ENAME is *non-transitive* since there is no set of attributes X where SSN \rightarrow X and X \rightarrow ENAME

Third Normal Form (2)

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

NOTE:

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

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

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

4 General Normal Form Definitions (For Multiple Keys) (1)

- The above definitions consider the primary key only
- The following more general definitions take into account relations with multiple candidate keys
- A relation schema R is in **second normal form** (2NF) if every non-prime attribute A in R is fully functionally dependent on *every key* of R

General Normal Form Definitions (2)

Definition:

- **Superkey** of relation schema R - a set of attributes S of R that contains a key of R
- A relation schema R is in **third normal form (3NF)** if whenever a FD $X \rightarrow A$ holds in R, then either:
 - (a) X is a superkey of R, or
 - (b) A is a prime attribute of R

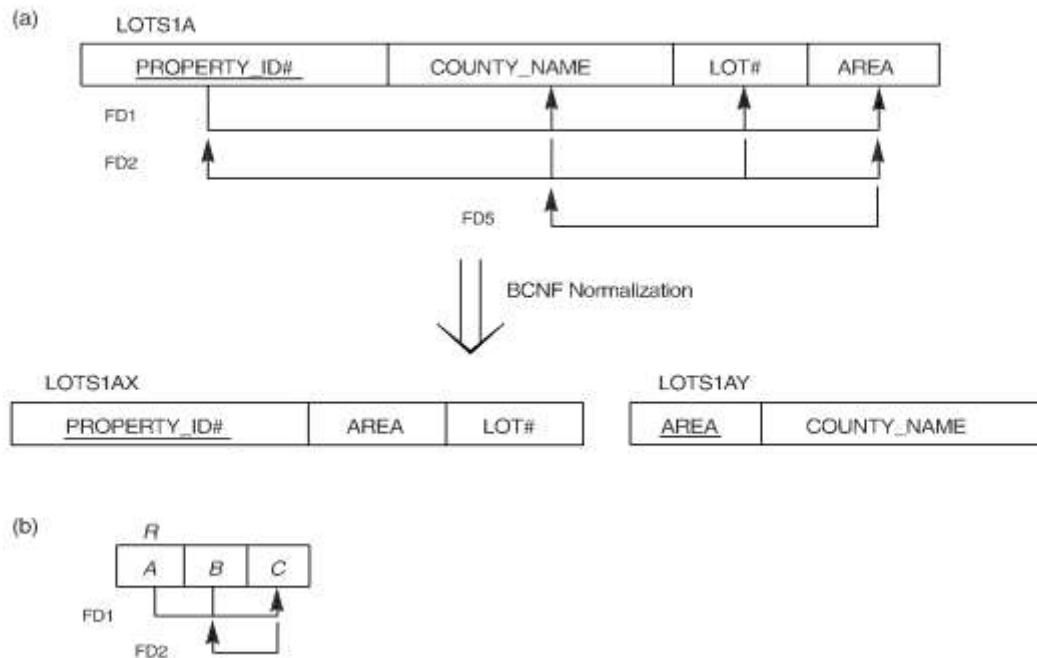
NOTE: Boyce-Codd normal form disallows condition (b) above

5 BCNF (Boyce-Codd Normal Form)

- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an FD $X \rightarrow A$ holds in R, then X is a superkey of R
- Each normal form is strictly stronger than the previous one
 - Every 2NF relation is in 1NF
 - Every 3NF relation is in 2NF
 - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- The goal is to have each relation in BCNF (or 3NF)

Figure 10.12 Boyce-Codd normal form

Figure 14.12 Boyce-Codd normal form. (a) BCNF normalization with the dependency of FD2 being “lost” in the decomposition. (b) A relation R in 3NF but not in BCNF.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.12 in Edition 4

Figure 10.13 a relation TEACH that is in 3NF but not in BCNF

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

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

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.13 in Edition 4

Achieving the BCNF by Decomposition (1)

- Two FDs exist in the relation TEACH:
fd1: { student, course } \rightarrow instructor
fd2: instructor \rightarrow course
- { student, course } is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 10.12 (b). So this relation is in 3NF but not in BCNF
- A relation **NOT** in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations. (See Algorithm 11.3)

Achieving the BCNF by Decomposition (2)

- Three possible decompositions for relation TEACH
 1. {student, instructor} and {student, course}
 2. {course, instructor} and {course, student}
 3. {instructor, course} and {instructor, student}
- All three decompositions will lose fd1. We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).
- A test to determine whether a binary decomposition (decomposition into two relations) is nonadditive (lossless) is discussed in section 11.1.4 under Property LJ1. Verify that the third decomposition above meets the property.

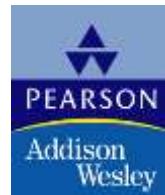
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 11

Relational Database Design Algorithms and Further Dependencies



Chapter Outline

0. Designing a Set of Relations
1. Properties of Relational Decompositions
2. Algorithms for Relational Database Schema
3. Multivalued Dependencies and Fourth Normal Form
4. Join Dependencies and Fifth Normal Form
5. Inclusion Dependencies
6. Other Dependencies and Normal Forms

DESIGNING A SET OF RELATIONS (1)

The Approach of Relational Synthesis (Bottom-up Design) :

- Assumes that all possible functional dependencies are known.
- First constructs a minimal set of FDs
- Then applies algorithms that construct a target set of 3NF or BCNF relations.
- Additional criteria may be needed to ensure the *the set of relations* in a relational database are satisfactory (see Algorithms 11.2 and 11.4).

DESIGNING A SET OF RELATIONS (2)

Goals:

- Lossless join property (a must) – algorithm 11.1 tests for general losslessness.
- Dependency preservation property – algorithms 11.3 decomposes a relation into BCNF components by sacrificing the dependency preservation.
- Additional normal forms
 - 4NF (based on multi-valued dependencies)
 - 5NF (based on join dependencies)

1. Properties of Relational Decompositions (1)

Relation Decomposition and Insufficiency of Normal Forms:

- **Universal Relation Schema:** a relation schema $R=\{A_1, A_2, \dots, A_n\}$ that includes all the attributes of the database.
- **Universal relation assumption:** every attribute name is unique.
- **Decomposition:** The process of decomposing the universal relation schema R into a set of relation schemas $D = \{R_1, R_2, \dots, R_m\}$ that will become the relational database schema by using the functional dependencies.

Properties of Relational Decompositions (2)

Relation Decomposition and Insufficiency of Normal Forms (cont.):

- **Attribute preservation condition:** Each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are “lost”.
- Another goal of decomposition is to have each individual relation R_i in the decomposition D be in BCNF or 3NF.
- Additional properties of decomposition are needed to prevent from generating spurious tuples

Properties of Relational Decompositions (3)

Dependency Preservation Property of a Decomposition :

Definition:

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

Properties of Relational Decompositions (4)

Dependency Preservation Property of a Decomposition (cont.):

- **Dependency Preservation Property:** a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each R_i in D is equivalent to F ; that is, $((\pi_{R1}(F)) \cup \dots \cup (\pi_{Rm}(F)))^+ = F^+$

(See examples in Fig 10.12a and Fig 10.11)

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

Properties of Relational Decompositions (5)

Lossless (Non-additive) Join Property of a Decomposition:

Definition:

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

$$*(\pi_{R1}(r), \dots, \pi_{Rm}(r)) = r$$

Note: The word loss in *lossless* refers to *loss of information*, not to loss of tuples. In fact, for “loss of information” a better term is “addition of spurious information”

Properties of Relational Decompositions (6)

Lossless (Non-additive) Join Property of a Decomposition (cont.):

Algorithm 11.1: Testing for Lossless 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.

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) *)

Properties of Relational Decompositions (7)

Lossless (Non-additive) Join Property of a Decomposition (cont.):

Algorithm 11.1: Testing for Lossless Join Property (cont.)

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 which 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 appear 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 lossless join property; otherwise it does not.

Properties of Relational Decompositions (8)

Lossless (nonadditive) join test for n -ary decompositions.

(a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test. (b) A decomposition of EMP_PROJ that has the lossless join property.

(a) $R=\{\text{SSN}, \text{ENAME}, \text{PNUMBER}, \text{PNAME}, \text{PLOCATION}, \text{HOURS}\}$ $D=\{R_1, R_2\}$

$$R_1 = \text{EMP_LOCS} = \{\text{ENAME}, \text{PLOCATION}\}$$

$$R_2 = \text{EMP_PROJ1} = \{\text{SSN}, \text{PNUMBER}, \text{HOURS}, \text{PNAME}, \text{PLOCATION}\}$$

$$F = \{\text{SSN} \rightarrow \text{ENAME}; \text{PNUMBER} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}; \{\text{SSN}, \text{PNUMBER}\} \rightarrow \text{HOURS}\}$$

	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

Properties of Relational Decompositions (8)

Lossless (nonadditive) join test for n -ary decompositions.

(c) Case 2:
Decomposition of
EMP_PROJ into EMP,
PROJECT, and
WORKS_ON satisfies
test.

(c) $R = \{\text{SSN}, \text{ENAME}, \text{PNUMBER}, \text{PNAME}, \text{PLOCATION}, \text{HOURS}\}$ $D = \{R_1, R_2, R_3\}$
 $R_1 = \text{EMP} = \{\text{SSN}, \text{ENAME}\}$
 $R_2 = \text{PROJ} = \{\text{PNUMBER}, \text{PNAME}, \text{PLOCATION}\}$
 $R_3 = \text{WORKS_ON} = \{\text{SSN}, \text{PNUMBER}, \text{HOURS}\}$

$F = \{\text{SSN} \rightarrow \{\text{ENAME}; \text{PNUMBER} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}; (\text{SSN}, \text{PNUMBER}) \rightarrow \text{HOURS}\}$

	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)

Properties of Relational Decompositions (9)

Testing Binary Decompositions for Lossless Join Property:

- **Binary Decomposition:** decomposition of a relation R into two relations.
- **PROPERTY LJ1 (lossless join test for binary decompositions):** 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 f.d. $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+ , or
 - The f.d. $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+ .

Properties of Relational Decompositions (10)

Successive Lossless Join Decomposition:

- **Claim 2 (Preservation of non-additivity in successive decompositions):**

If a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R has the lossless (non-additive) join property with respect to a set of functional dependencies F on R , and if a decomposition $D_i = \{Q_1, Q_2, \dots, Q_k\}$ of R_i has the lossless (non-additive) join property with respect to the projection of F on R_i , then the decomposition $D_2 = \{R_1, R_2, \dots, R_{i-1}, Q_1, Q_2, \dots, Q_k, R_{i+1}, \dots, R_m\}$ of R has the non-additive join property with respect to F .

2. Algorithms for Relational Database Schema Design (1)

Algorithm 11.2: Relational Synthesis into 3NF with Dependency Preservation (*Relational Synthesis Algorithm*)

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

1. Find a minimal cover G for F (use Algorithm 10.2);
2. For each left-hand-side X of a functional dependency that appears in G , create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the *key* of this relation) ;
3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.

Claim 3: Every relation schema created by Algorithm 11.2 is in 3NF.

Algorithms for Relational Database Schema Design (2)

Algorithm 11.3: Relational Decomposition into BCNF with Lossless (non-additive) join property

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

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

Assumption: *No null values are allowed for the join attributes.*

Algorithms for Relational Database Schema Design (3)

Algorithm 11.4 Relational Synthesis into 3NF with Dependency Preservation and Lossless (Non-Additive) Join Property

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

1. Find a minimal cover G for F (*Use Algorithm 10.2*).
2. For each left-hand-side X of a functional dependency that appears in G , create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the *key* of this relation).
3. If none of the relation schemas in D contains a key of R , then create one more relation schema in D that contains attributes that form a key of R . (*Use Algorithm 11.4a to find the key of R*)

Algorithms for Relational Database Schema Design (4)

Algorithm 11.4a Finding a Key K for R Given a set F of Functional Dependencies

Input: A universal 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 $K := K - \{A\}$; }

Algorithms for Relational Database Schema Design (5)

Issues with null-value joins. (a) Some EMPLOYEE tuples have null for the join attribute

DNUM.

(a)

EMPLOYEE

ENAME	SSN	BDATE	ADDRESS	DNUM
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	null
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	null

DEPARTMENT

DNAME	DNUM	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

Algorithms for Relational Database Schema Design (5)

Issues with null-value joins. (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations. (c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

(b)

ENAME	SSN	BDATE	ADDRESS	DNUM	DNAME	DMGRSSN
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 Fire Oak, 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

(c)

ENAME	SSN	BDATE	ADDRESS	DNUM	DNAME	DMGRSSN
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 Fire Oak, 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
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	null	null	null
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	null	null	null

Algorithms for Relational Database Schema Design (6)

The “dangling tuple” problem. (a) The relation EMPLOYEE_1 (includes all attributes of EMPLOYEE from figure 11.2a except DNUM).

(a) **EMPLOYEE_1**

ENAME	SSN	BDATE	ADDRESS
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX

Algorithms for Relational Database Schema Design (6)

The “dangling tuple” problem. (b) The relation EMPLOYEE_2 (includes DNUM attribute with null values). (c) The relation EMPLOYEE_3 (includes DNUM attribute but does not include tuples for which DNUM has null values).

(b) **EMPLOYEE_2**

SSN	DNUM
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	null
888664444	null

(c) **EMPLOYEE_3**

SSN	DNUM
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

Algorithms for Relational Database Schema Design (7)

Discussion of Normalization Algorithms:

Problems:

- The database designer must first specify *all* the relevant functional dependencies among the database attributes.
- These algorithms are *not deterministic* in general.
- It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF (instead of 3NF as in Algorithm 11.4).

Algorithms for Relational Database Schema Design (8)

Table 11.1 Summary of some of the algorithms discussed above

Algorithm	Input	Output	Properties/Purpose	Remarks
11.1	A decomposition D of R and a set F of functional dependencies	Boolean result: yes or no for lossless join property	Testing for non-additive join decomposition	See a simpler test in Section 11.1.4 for binary decompositions
11.2	Set of functional dependencies F	A set of relations in 3NF	Dependency preservation	No guarantee of satisfying lossless join property
11.3	Set of functional dependencies F	A set of relations in BCNF	Lossless join decomposition	No guarantee of dependency preservation
11.4	Set of functional dependencies F	A set of relations in 3NF	Lossless join and dependency preserving decomposition	May not achieve BCNF
11.4a	Relation schema R with a set of functional dependencies F	Key K of R	To find a key K (which is a subset of R)	The entire relation R is always a default superkey

3. Multivalued Dependencies and Fourth Normal Form (1)

(a) The EMP relation with two MVDs: ENAME $\rightarrow\!\!\!\rightarrow$ PNAME and ENAME $\rightarrow\!\!\!\rightarrow$ DNAME. (b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and

EMP_DEPENDENTS.

(a) **EMP**

ENAME	PNAME	DNAME
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) **EMP_PROJECTS**

ENAME	PNAME
Smith	X
Smith	Y

EMP_DEPENDENTS

ENAME	DNAME
Smith	John
Smith	Anna

3. Multivalued Dependencies and Fourth Normal Form (1)

- (c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R1, R2, R3).
(d) Decomposing the relation SUPPLY into the 5NF relations R1, R2, and R3.

(c) **SUPPLY**

SNAME	PARTNAME	PROJNAME
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) **R1**

SNAME	PARTNAME
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

SNAME	PROJNAME
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R3

PARTNAME	PROJNAME
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Multivalued Dependencies and Fourth Normal Form (2)

Definition:

- A **multivalued dependency (MVD)** $X \rightarrow\!\!> Y$ 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 $(R \setminus (X \cup Y))$:
 - $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
 - $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
 - $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.
- An MVD $X \rightarrow\!\!> Y$ in R is called a **trivial MVD** if (a) Y is a subset of X , or (b) $X \cup Y = R$.

Multivalued Dependencies and Fourth Normal Form (3)

Inference Rules for Functional and Multivalued Dependencies:

IR1 (reflexive rule for FDs): If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule for FDs): $\{X \rightarrow Y\} \mid = XZ \rightarrow YZ$.

IR3 (transitive rule for FDs): $\{X \rightarrow Y, Y \rightarrow Z\} \mid = X \rightarrow Z$.

IR4 (complementation rule for MVDs): $\{X \multimap Y\} \mid = X \multimap (R - (X \cup Y))$.

IR5 (augmentation rule for MVDs): If $X \multimap Y$ and $W \supseteq Z$ then $WX \multimap YZ$.

IR6 (transitive rule for MVDs): $\{X \multimap Y, Y \multimap Z\} \mid = X \multimap (Z \Delta Y)$.

IR7 (replication rule for FD to MVD): $\{X \rightarrow Y\} \mid = X \multimap Y$.

IR8 (coalescence rule for FDs and MVDs): If $X \multimap Y$ and there exists W with the properties that (a) $W \cap Y$ is empty, (b) $W \rightarrow Z$, and (c) $Y \supseteq Z$, then $X \rightarrow Z$.

Multivalued Dependencies and Fourth Normal Form (4)

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 $X \multimap\multimap Y$ in F^+ , X is a superkey for R .

Note: F^+ is the (complete) set of all dependencies (functional or multivalued) that will hold in every relation state r of R that satisfies F . It is also called the **closure** of F .

Multivalued Dependencies and Fourth Normal Form (5)

Decomposing a relation state of EMP that is not in 4NF. (a) EMP relation with additional tuples. (b) Two corresponding 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

(a) **EMP**

ENAME	PNAME	DNAME
-------	-------	-------

Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) **EMP_PROJECTS**

ENAME	PNAME
-------	-------

Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

ENAME	DNAME
-------	-------

Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

Multivalued Dependencies and Fourth Normal Form (6)

Lossless (Non-additive) Join Decomposition into 4NF Relations:

- **PROPERTY LJ1'**

The relation schemas R_1 and R_2 form a lossless (non-additive) join decomposition of R with respect to a set F of functional *and* multivalued dependencies if and only if

$$(R_1 \cap R_2) \longrightarrow\!\!\!> (R_1 - R_2)$$

or by symmetry, if and only if

$$(R_1 \cap R_2) \longrightarrow\!\!\!> (R_2 - R_1)).$$

Multivalued Dependencies and Fourth Normal Form (7)

Algorithm 11.5: Relational decomposition into 4NF relations with non-additive join property

Input: A universal relation R and a set of functional and multivalued dependencies F.

1. Set $D := \{ R \}$;
2. While there is a relation schema Q in D that is not in 4NF do
 - { choose a relation schema Q in D that is not in 4NF;
find a nontrivial MVD $X \multimap\!\!> Y$ in Q that violates 4NF;
replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
 - };

4. Join Dependencies and Fifth Normal Form (1)

Definition:

- 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 non-additive join decomposition into R_1, R_2, \dots, R_n ; that is, for every such r we have

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

Note: an MVD is a special case of a JD where n = 2.

- 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 .

Join Dependencies and Fifth Normal Form (2)

Definition:

- 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^+ (that is, implied by F), every R_i is a superkey of R .

Relation SUPPLY with Join Dependency and conversion to Fifth Normal Form

- (c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R1, R2, R3).
(d) Decomposing the relation SUPPLY into the 5NF relations R1, R2, and R3.

(c) **SUPPLY**

SNAME	PARTNAME	PROJNAME
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) **R1**

SNAME	PARTNAME
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

SNAME	PROJNAME
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R3

PARTNAME	PROJNAME
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

5. Inclusion Dependencies (1)

Definition:

- An **inclusion dependency** $R.X < S.Y$ between two sets of attributes— X of relation schema R , and Y of relation schema S —specifies the constraint that, at any specific time when r is a relation state of R and s a relation state of S , we must have

$$\pi_X(r(R)) \supseteq \pi_Y(s(S))$$

Note: The ? (subset) relationship does not necessarily have to be a proper subset. The sets of attributes on which the inclusion dependency is specified— X of R and Y of S —must have the same number of attributes. In addition, the domains for each pair of corresponding attributes should be compatible.

Inclusion Dependencies (2)

Objective of Inclusion Dependencies:

To formalize two types of interrelational constraints which cannot be expressed using F.D.s or MVDs:

- Referential integrity constraints
- Class/subclass relationships

● Inclusion dependency inference rules

IDIR1 (reflexivity): $R.X < R.X$.

IDIR2 (attribute correspondence): If $R.X < S.Y$

where $X = \{A_1, A_2, \dots, A_n\}$ and $Y = \{B_1, B_2, \dots, B_n\}$ and A_i Corresponds-to B_i , then $R.A_i < S.B_i$ for $1 \leq i \leq n$.

IDIR3 (transitivity): If $R.X < S.Y$ and $S.Y < T.Z$, then $R.X < T.Z$.

6. Other Dependencies and Normal Forms (1)

Template Dependencies:

- Template dependencies provide a technique for representing constraints in relations that typically have no easy and formal definitions.
- The idea is to specify a template—or example—that defines each constraint or dependency.
- There are two types of templates: tuple-generating templates and constraint-generating templates.
- A template consists of a number of **hypothesis tuples** that are meant to show an example of the tuples that may appear in one or more relations. The other part of the template is the **template conclusion**.

Other Dependencies and Normal Forms (2)

Templates for some common types of dependencies.

(a) Template for functional dependency $X \rightarrow Y$.

(b) Template for the multivalued dependency $X \rightarrow\!> Y$.

(c) Template for the inclusion dependency $R.X < S.Y$.

(a)

	$R = \{ A, B, C, D \}$	
hypothesis	$\begin{array}{ccc} a_1 & b_1 & c_1 \\ a_1 & b_1 & c_2 \end{array}$	$X = \{ A, B \}$
conclusion	<hr/>	$Y = \{ C, D \}$
	$c_1 = c_2 \text{ and } d_1 = d_2$	

(b)

	$R = \{ A, B, C, D \}$	
hypothesis	$\begin{array}{cccc} a_1 & b_1 & 1 & d_1 \\ a_1 & b_1 & 2 & d_2 \end{array}$	$X = \{ A, B \}$
conclusion	<hr/> $\begin{array}{cccc} a_1 & b_1 & 2 & d_1 \\ a_1 & b_1 & 1 & d_2 \end{array}$	$Y = \{ C \}$

(c)

	$R = \{ A, B, C, D \}$	$S = \{ E, F, G \}$
hypothesis	$\begin{array}{cccc} a_1 & b_1 & c_1 & d_1 \end{array}$	$X = \{ C, D \}$
conclusion	<hr/>	$Y = \{ E, F \}$
	$c_1 \quad d_1 \quad g$	

Other Dependencies and Normal Forms (3)

Templates for the constraint that an employee's salary must be less than the supervisor's salary.

EMPLOYEE = { NAME , SSN , ... , SALARY , SUPERVISORSSN }

	a	b	c	d
hypothesis	e	d	f	g
<hr/>				
conclusion			c < f	

Other Dependencies and Normal Forms (4)

Domain-Key Normal Form (DKNF):

- **Defintion:** A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation.
- The **idea** is to specify (theoretically, at least) the “*ultimate normal form*” that takes into account all possible types of dependencies and constraints. .
- For a relation in DKNF, it becomes very straightforward to enforce all database constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.
- The practical utility of DKNF is limited

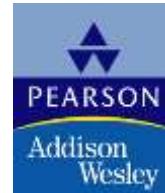
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 12

Practical Database Design Methodology and Use of UML Diagrams



Copyright © 2004 Pearson Education, Inc.

The Role of Information Systems in Organizations

- The Organizational Context for Using Database Systems
- The Information System Life Cycle
- The Database Application System Life Cycle

The Database Design and Implementation Process

- Phase 1: Requirements Collection and Analysis
- Phase 2: Conceptual Database Design
- Phase 3: Choice of DBMS
- Phase 4: Data Model Mapping (Logical Database Design)
- Phase 5: Physical Database Design
- Phase 6: Database System Implementation and Tuning

Use of UML Diagrams as an Aid to Database Design Specification

- UML As a Design Specification Standard
- UML for Database Application Design
- Different Diagrams in UML
- A Modeling and Design Example:
University Database

Relational Rose, A UML Based Design Tool

- Relational Rose for Database Design
- Relational Rose Data Modeler
- Data Modeling Using Rational Rose Data Modeler

Automated Database Design Tools

Summary

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 13

Disk Storage, Basic File Structures, and Hashing.



Chapter Outline

- Disk Storage Devices
- Files of Records
- Operations on Files
- Unordered Files
- Ordered Files
- Hashed Files
 - Dynamic and Extendible Hashing Techniques
- RAID Technology

Disk Storage Devices (cont.)

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A *disk pack* contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular *tracks* on each disk *surface*. Track capacities vary typically from 4 to 50 Kbytes.

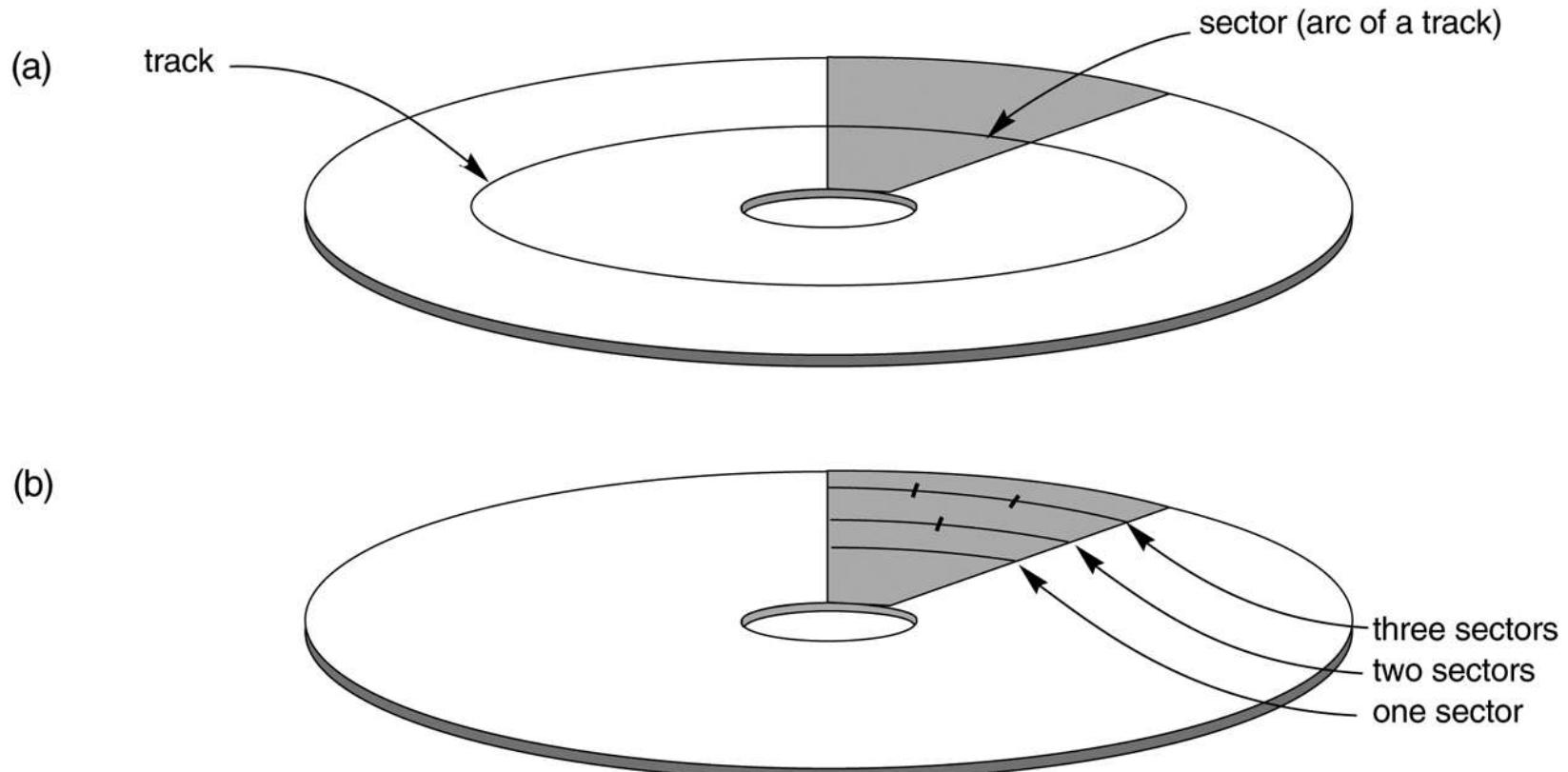
Disk Storage Devices (cont.)

Because a track usually contains a large amount of information, it is divided into smaller *blocks* or *sectors*.

- The division of a track into *sectors* is hard-coded on the disk surface and cannot be changed. One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector.
- A track is divided into *blocks*. The block size B is fixed for each system. Typical block sizes range from $B=512$ bytes to $B=4096$ bytes.

Whole blocks are transferred between disk and

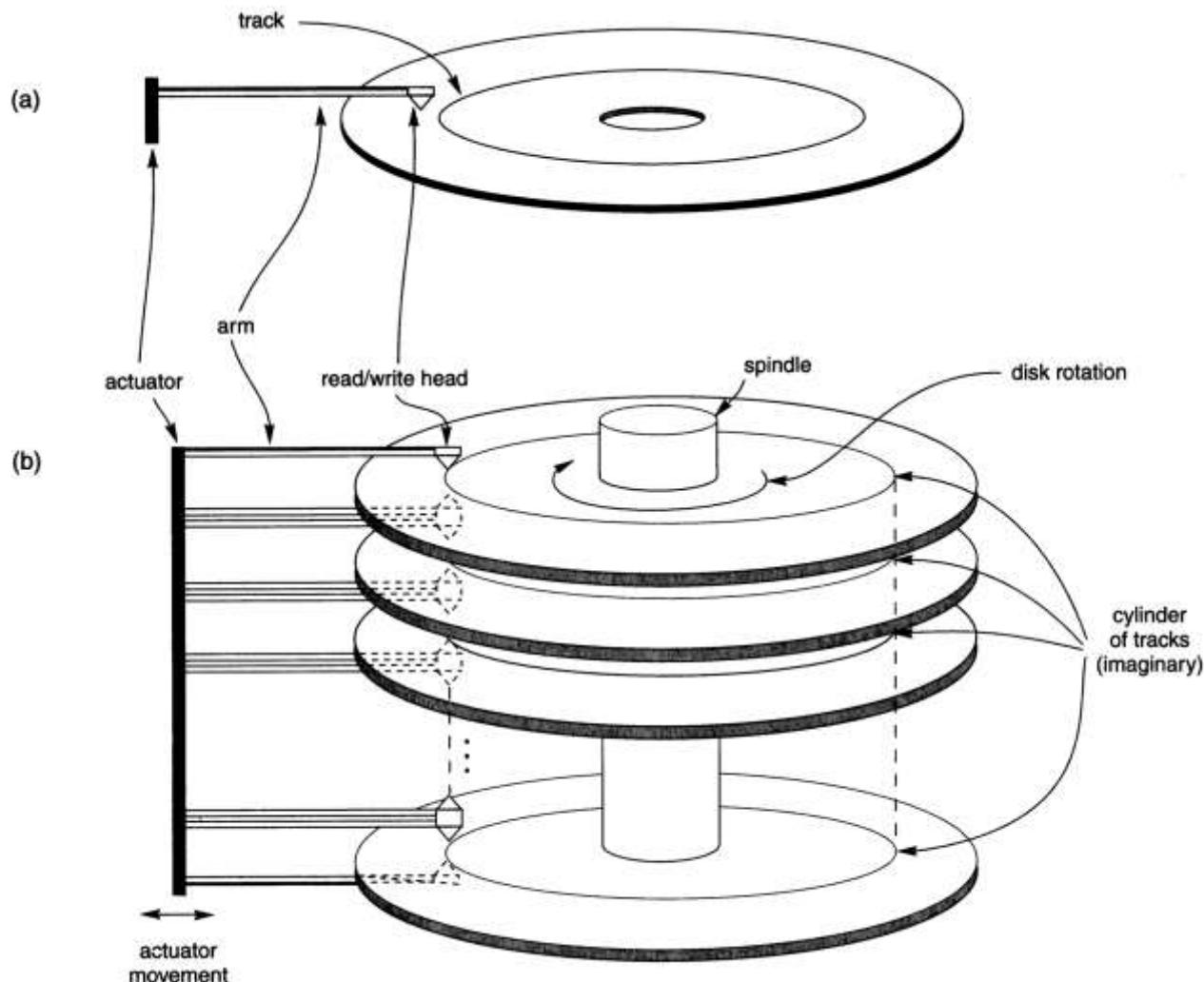
Disk Storage Devices (cont.)



Disk Storage Devices (cont.)

- A *read-write* head moves to the track that contains the block to be transferred. Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces), the track number or surface number (within the cylinder), and block number (within track).
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) rd .
- Double buffering can be used to speed up the transfer of contiguous disk blocks.

Disk Storage Devices (cont.)



Typical Disk Parameter S

TABLE 13.1 SPECIFICATIONS OF TYPICAL HIGH-END CHEETAH DISKS FROM SEAGATE

Description	Cheetah X15 36LP	Cheetah 10K.6
Model Number	ST336732LC	ST3146807LC
Form Factor (width)	3.5 inch	3.5 inch
Height	25.4 mm	25.4 mm
Width	101.6 mm	101.6 mm
Weight	0.68 Kg	0.73 Kg
Capacity/Interface		
Formatted Capacity	36.7 Gbytes	146.8 Gbytes
Interface Type	80-pin	80-pin
Configuration		
Number of disks (physical)	4	4
Number of heads (physical)	8	8
Number of Cylinders	18,479	49,854
Bytes per Sector	512	512
Areal Density	N/A	36,000 Mbits/sq.inch
Track Density	N/A	64,000 Tracks/inch
Recording Density	N/A	570,000 bits/inch
Performance		
Transfer Rates		
Internal Transfer Rate (min)	522 Mbits/sec	475 Mbits/sec
Internal Transfer Rate (max)	709 Mbits/sec	840 Mbits/sec
Formatted Int. Transfer Rate (min)	51 MBytes/sec	43 MBytes/sec
Formatted Int. Transfer Rate (max)	69 MBytes/sec	78 MBytes/sec
External I/O Transfer Rate (max)	320 MBytes/sec	320 MBytes/sec
Seek Times		
Avg. Seek Time (Read)	3.6 msec (typical)	4.7 msec (typical)
Avg. Seek Time (Write)	4.2 msec (typical)	5.2 msec (typical)
Track-to-track Seek, Read	0.5 msec (typical)	0.3 msec (typical)
Track-to-track Seek, Write	0.8 msec (typical)	0.5 msec (typical)
Average Latency	2 msec	2.99 msec
Other		
Default Buffer (cache) size	8,192 Kbytes	8,000 Kbytes
Spindle Speed	15K rpm	10K rpm

Records

- Fixed and variable length records
- Records contain fields which have values of a particular type (e.g., amount, date, time, age)
- Fields themselves may be fixed length or variable length
- Variable length fields can be mixed into one record: separator characters or length fields are needed so that the record can be

Blocking

- Blocking: refers to storing a number of records in one block on the disk.
- Blocking factor (bfr) refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.
- *Spanned Records*: refer to records that exceed the size of one or more blocks and

Files of Records

- A file is a *sequence* of records, where each record is a collection of data values (or data items).
- A *file descriptor* (or *file header*) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.
- Records are stored on disk blocks. The *blocking factor bfr* for a file is the (average) number of file records stored in a disk block.

Files of Records (cont.)

- File records can be *unspanned* (no record can span two blocks) or *spanned* (a record can be stored in more than one block).
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous*, *linked*, or *indexed*.
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.
- Files of variable-length records require additional information to be stored in each

Operation on Files

Typical file operations include:

- **OPEN:** Readies the file for access, and associates a pointer that will refer to a *current* file record at each point in time.
- **FIND:** Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- **FINDNEXT:** Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- **READ:** Reads the current file record into a program variable.

Operation on Files (cont.)

- **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- **MODIFY:** Changes the values of some fields of the current file record.
- **CLOSE:** Terminates access to the file.
- **REORGANIZE:** Reorganizes the file records.
For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **READ_ORDERED:** Read the file blocks in order of a specific field of the file.

Unordered Files

- Also called a *heap* or a *pile* file.
- New records are inserted at the end of the file.
- To search for a record, a *linear search* through the file records is necessary. This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.

Ordered Files

- Also called a *sequential file*.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the *correct order*. It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A *binary search* can be used to search for a record on its *ordering field value*. This requires reading and searching \log_2 of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field

Ordered Files (cont.)

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
		⋮				
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
		⋮				
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
		⋮				
	Alien, Sam					
block 4	Alien, Troy					
	Anders, Keith					
		⋮				
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
		⋮				
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
		⋮				
	Atkins, Timothy					
	⋮					
block n - 1	Wong, James					
	Wood, Donald					
		⋮				
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
		⋮				
	Zimmer, Byron					

Average Access Times

The following table shows the average access time to access a specific record for a given type of file

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

Hashed Files

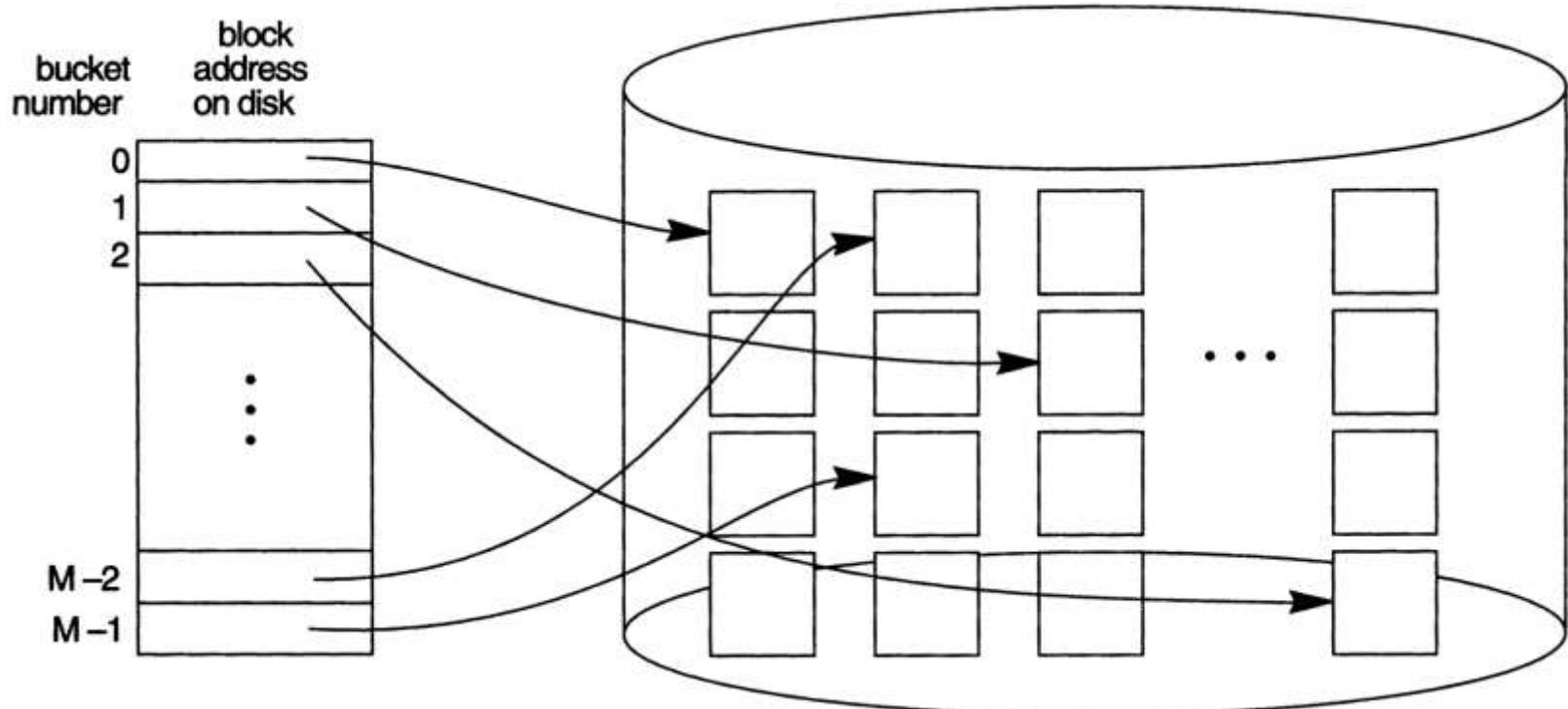
- Hashing for disk files is called *External Hashing*
- The file blocks are divided into M equal-sized *buckets*, numbered $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$. Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the hash key of the file.
- The record with hash key value K is stored in bucket i, where $i=h(K)$, and h is the *hashing function*.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full. An overflow file is kept for storing such records. Overflow records that hash to each bucket can be linked together.

Hashed Files (cont.)

There are numerous methods for collision resolution, including the following:

- ***Open addressing:*** Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
- ***Chaining:*** For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
- ***Multiple hashing:*** The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

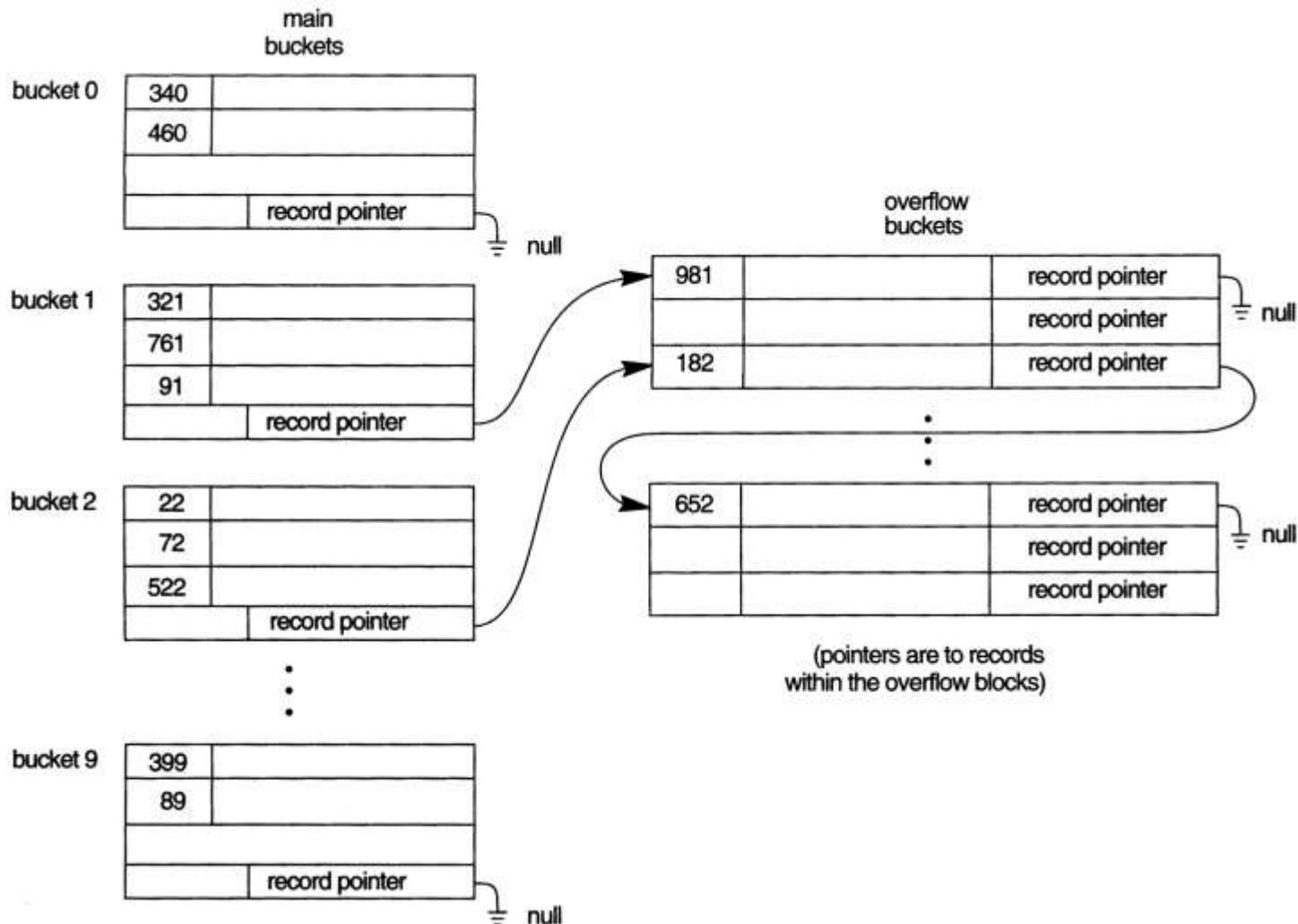
Hashed Files (cont.)



Hashed Files (cont.)

- To reduce overflow records, a hash file is typically kept 70-80% full.
- The hash function h should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of static external hashing:
 - Fixed number of buckets M is a problem if the number of records in the file grows or shrinks.
 - Ordered access on the hash key is quite inefficient (requires sorting the records)

Hashed Files - Overflow handling



Dynamic And Extendible Hashed Files

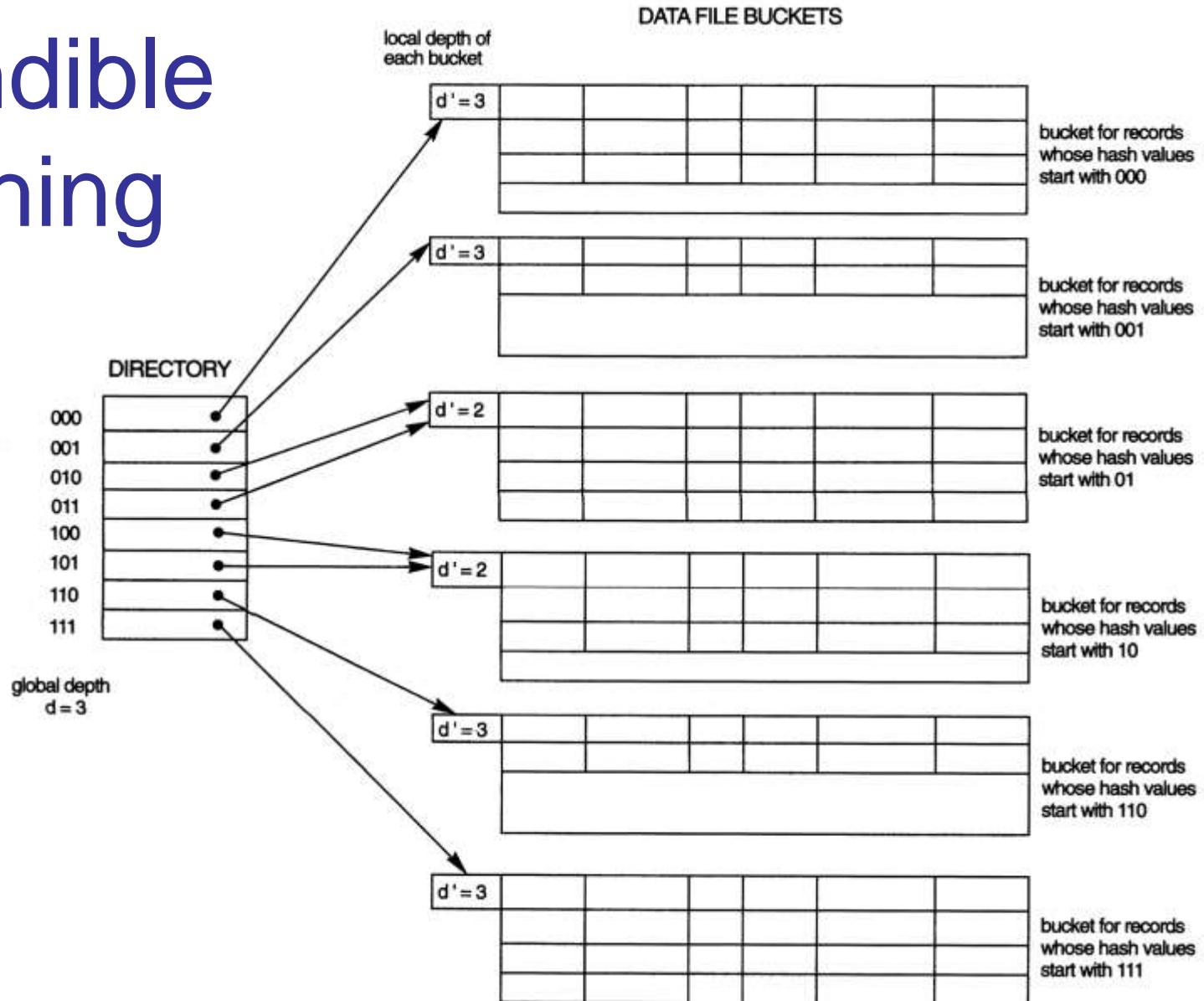
Dynamic and Extendible Hashing Techniques

- Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.
- These techniques include the following:
dynamic hashing , *extendible hashing* , and
linear hashing .
- Both dynamic and extendible hashing use the *binary representation* of the hash value $h(K)$ in order to access a *directory*. In dynamic hashing

Dynamic And Extendible Hashing (cont.)

- The directories can be stored on disk, and they expand or shrink dynamically. Directory entries point to the disk blocks that contain the stored records.
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks. The directory is updated appropriately.
- Dynamic and extendible hashing do not require an overflow area.
- Linear hashing does require an overflow area but does not use a directory. Blocks are split in *linear order* as the file expands.

Extendible Hashing

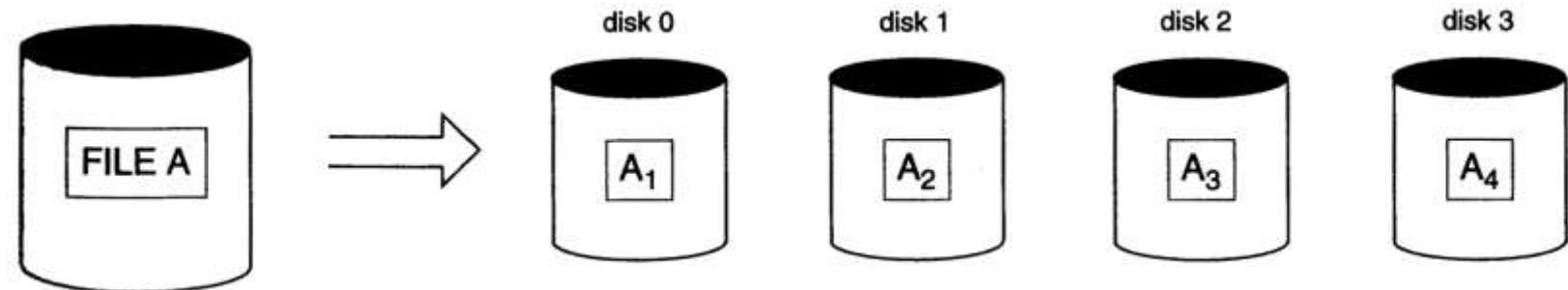


Parallelizing Disk Access using RAID Technology.

- Secondary storage technology must take steps to keep up in performance and reliability with processor technology.
- A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**.
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.

RAID Technology (cont.)

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



RAID Technology (cont.)

Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.

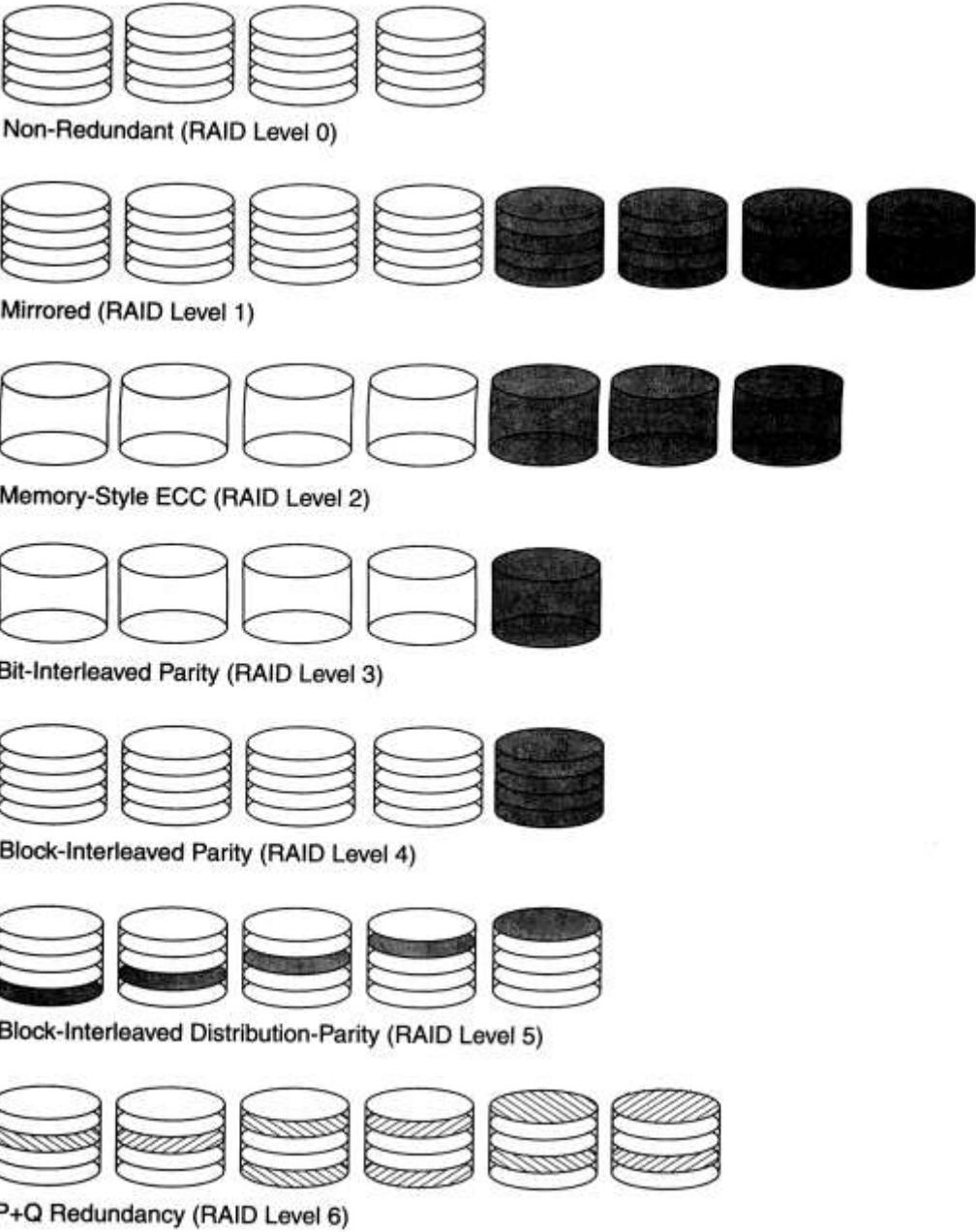
- Raid level 0 has no redundant data and hence has the best write performance.
- Raid level 1 uses mirrored disks.
- Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
- Raid level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed.
- Raid Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks.
- Raid level 6 applies the so-called $P + Q$ redundancy scheme using Reed-Solomon codes to protect against up to two disk failures by using just two redundant disks.

Use of RAID Technology (cont.)

Different raid organizations are being used under different situations

- Raid level 1 (mirrored disks) is the easiest for rebuild of a disk from other disks
 - It is used for critical applications like logs
- Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
- Raid level 3 (single parity disks relying on the disk controller to figure out which disk has failed) and level 5 (block-level data striping) are preferred for Large volume storage, with level 3 giving higher transfer rates.
- Most popular uses of the RAID technology currently are: Level 0 (with striping), Level 1 (with mirroring) and Level 5 with an extra drive for parity.
- Design Decisions for RAID include – level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping.

Use of RAID Technology (cont.)



Trends in Disk Technology

TABLE 13.3 TRENDS IN DISK TECHNOLOGY

	1993 PARAMETER VALUES*	HISTORICAL RATE OF IMPROVEMENT PER YEAR (%)*	CURRENT (2003) VALUES**
Areal density	50–150 Mbits/sq. inch	27	36 Gbits/sq. inch
Linear density	40,000–60,000 bits/inch	13	570 Kbits/inch
Inter-track density	1500–3000 tracks/inch	10	64,000 tracks/inch
Capacity (3.5" form factor)	100–2000 MB	27	146 GB
Transfer rate	3–4 MB/s	22	43–78 MB/sec
Seek time	7–20 ms	8	3.5–6 msec

*Source: From Chen, Lee, Gibson, Katz, and Patterson (1994), *ACM Computing Surveys*, Vol. 26, No. 2 (June 1994). Reprinted by permission.

**Source: IBM Ultrastar 36XP and 18ZX hard disk drives.

Storage Area Networks

- The demand for higher storage has risen considerably in recent times.
- Organizations have a need to move from a static fixed data center oriented operation to a more flexible and dynamic infrastructure for information processing.
- Thus they are moving to a concept of Storage Area Networks (SANs). In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.
- This allows storage systems to be placed at longer distances from the servers and provide

Storage Area Networks (contd.)

Advantages of SANs are:

- Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches.
- Up to 10km separation between a server and a storage system using appropriate fiber optic cables.
- Better isolation capabilities allowing nondisruptive addition of new peripherals and servers.
- SANs face the problem of combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware.

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 14

Indexing Structures for Files



Chapter Outline

- Types of Single-level Ordered Indexes
 - Primary Indexes
 - Clustering Indexes
 - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys

Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries **<field value, pointer to record>**, which is ordered by field value
- The index is called an *access path* on the field.

Indexes as Access Paths (contd.)

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse.
 - A **dense index** has an index entry for *every search key value* (and hence every record) in the data file.
 - A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values

Indexes as Access Paths (contd.)

Example: Given the following data file:

EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)

Suppose that:

record size $R=150$ bytes

block size $B=512$ bytes

$r=30000$ records

Then, we get:

blocking factor $Bfr = B \text{ div } R = 512 \text{ div } 150 = 3$ records/block

number of file blocks $b = (r/Bfr) = (30000/3) = 10000$ blocks

For an index on the SSN field, assume the field size $V_{SSN}=9$ bytes,

assume the record pointer size $P_R=7$ bytes. Then:

index entry size $R_I=(V_{SSN}+ P_R)=(9+7)=16$ bytes

index blocking factor $Bfr_I = B \text{ div } R_I = 512 \text{ div } 16 = 32$ entries/block

number of index blocks $b = (r/Bfr_I) = (30000/32) = 938$ blocks

binary search needs $\log_2 b = \log_2 30000 = 10$ block accesses

This is compared to an average linear search cost of:

$(b/2) = 30000/2 = 15000$ block accesses

If the file records are ordered, the binary search cost would be:

$\log_2 b = \log_2 30000 = 15$ block accesses

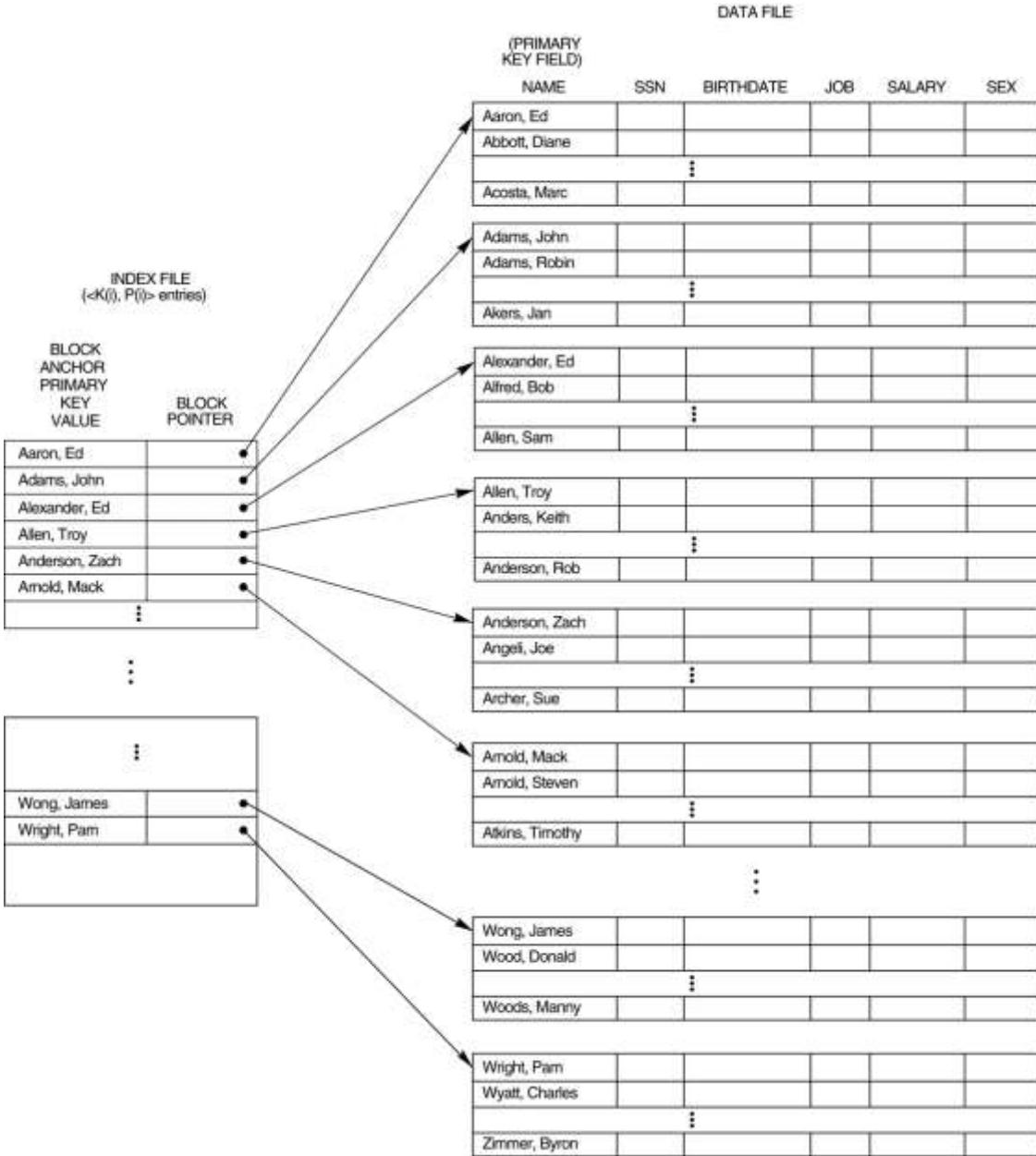
Types of Single-Level Indexes

● Primary Index

- Defined on an ordered data file
- The data file is ordered on a *key field*
- Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
- A similar scheme can use the *last record* in a block.
- A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

FIGURE 14.1

Primary index on the ordering key field of the file shown in Figure 13.7.



Types of Single-Level Indexes

● Clustering Index

- Defined on an ordered data file
- The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
- Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
- It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.

(CLUSTERING FIELD)

DEPTNUMBER NAME SSN JOB BIRTHDATE SALARY

1					
1					
1					
2					

2					
3					
3					
3					

3					
3					
4					
4					

5					
5					
5					
5					

6					
6					
6					
6					

6					
8					
8					
8					

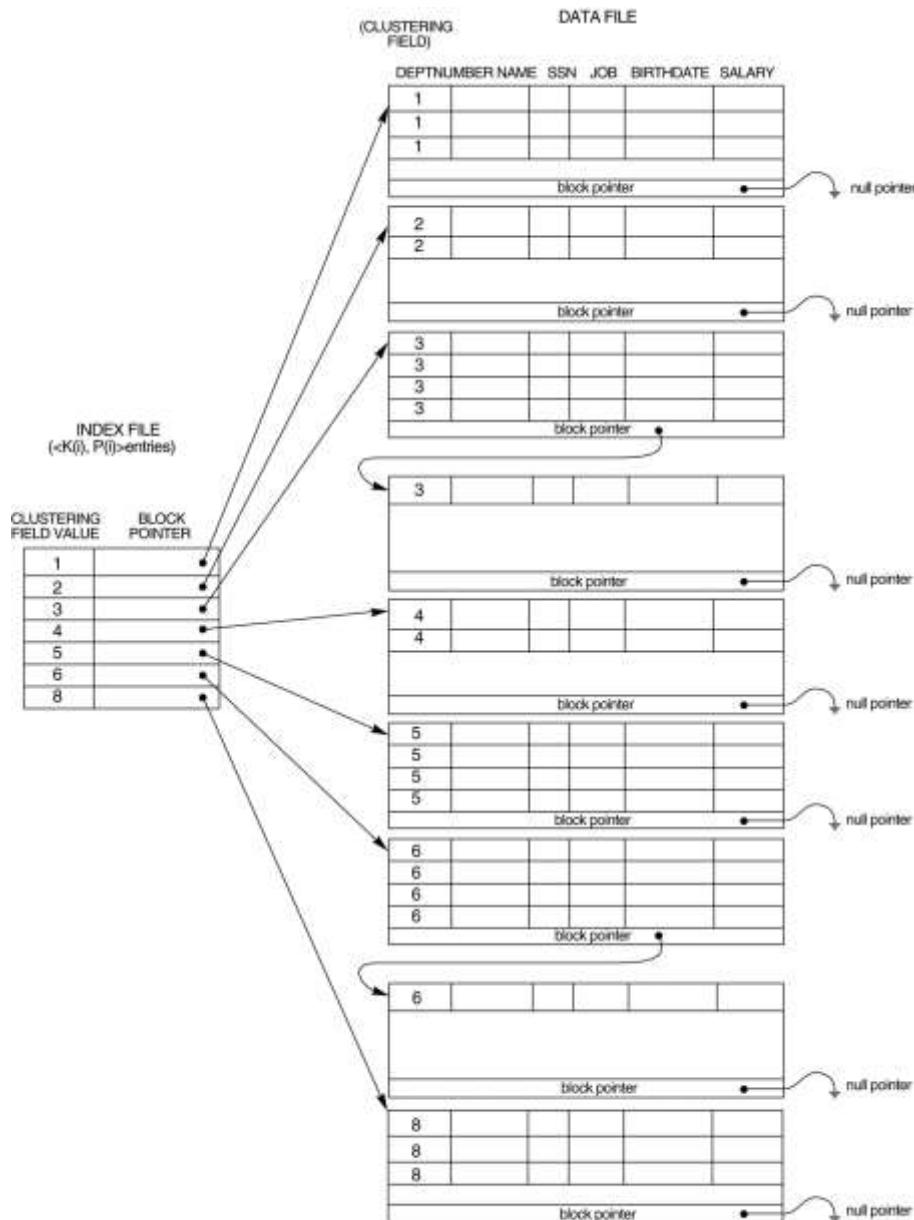
INDEX FILE
($<K(i), P(i)>$ entries)

CLUSTERING FIELD VALUE BLOCK POINTER

1	
2	
3	
4	
5	
6	
8	

FIGURE 14.2
A clustering index on the
DEPTNUMBER ordering nonkey
field of an EMPLOYEE file.

FIGURE 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Types of Single-Level Indexes

● Secondary Index

- A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- The secondary index may be on a field which is a candidate key and has a unique value in every record, or a nonkey with duplicate values.
- The index is an ordered file with two fields.
 - The first field is of the same data type as some *nonordering field* of the data file that is an *indexing field*.
 - The second field is either a *block* pointer or a *record* pointer. There can be *many* secondary indexes (and hence, indexing fields) for the same file.
- Includes one entry *for each record* in the data file; hence, it is a *dense index*

FIGURE 14.4
 A dense
 secondary index
 (with block
 pointers) on a
 nonordering key
 field of a file.

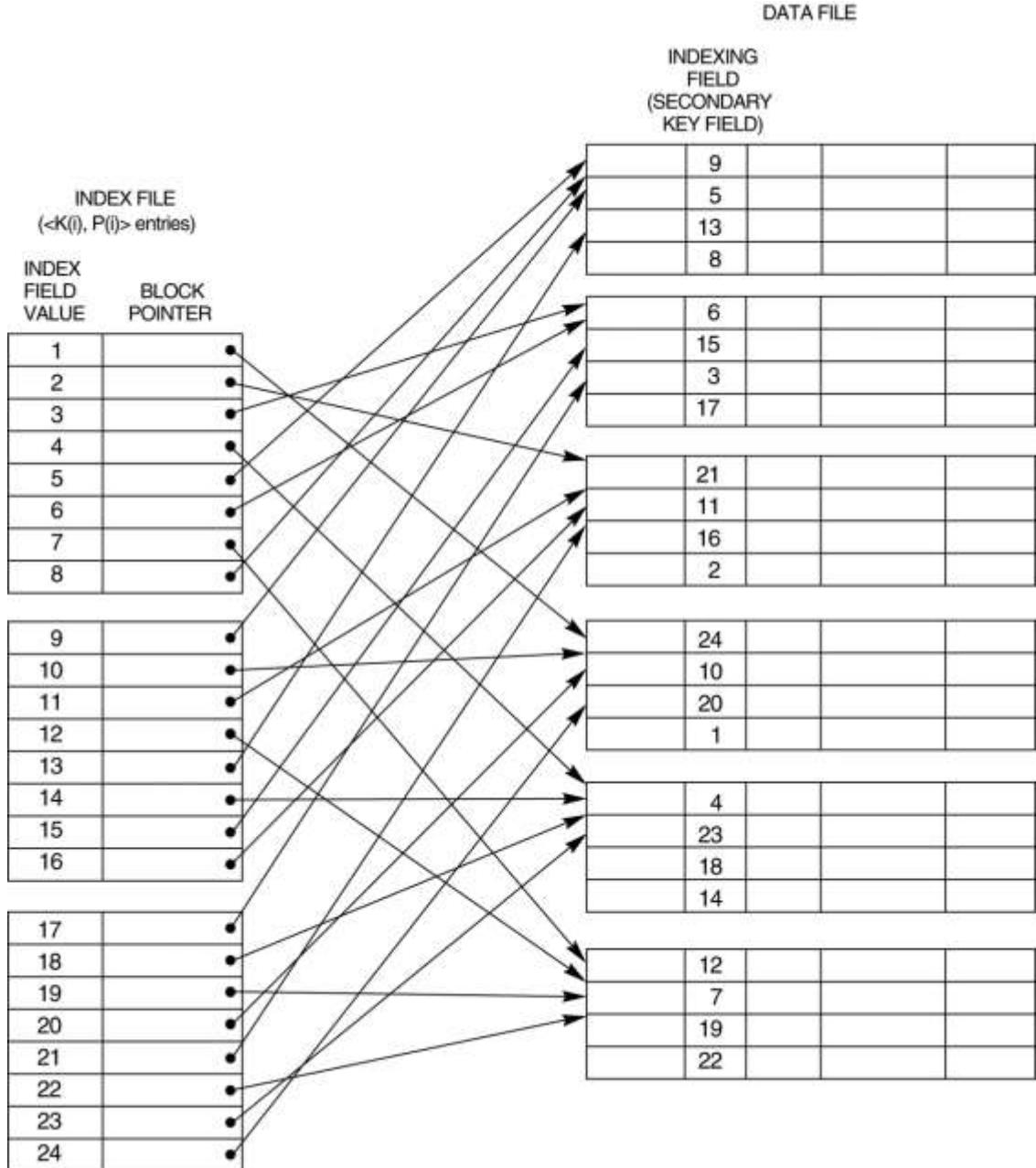


FIGURE 14.5

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

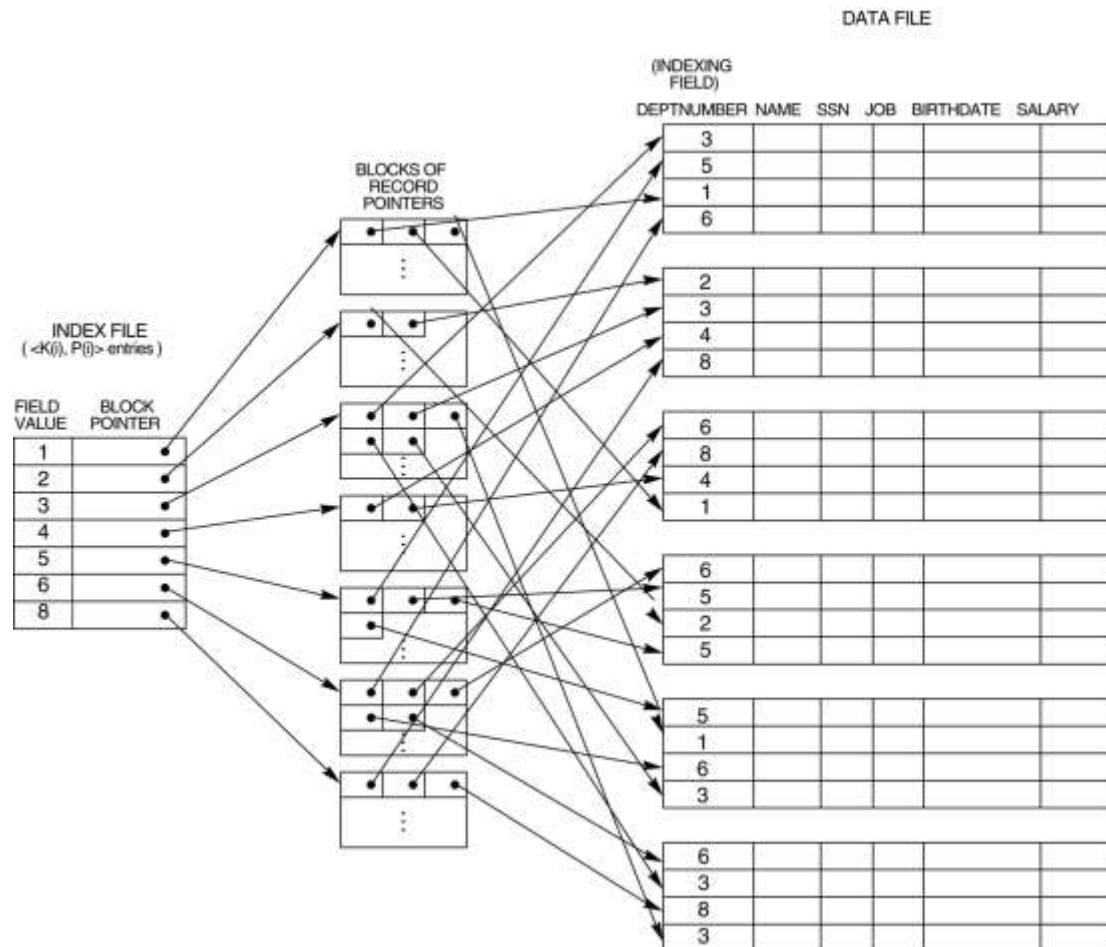


TABLE 14.2 PROPERTIES OF INDEX TYPES

Type of Index	Number of (First-level) Index Entries	Dense or Nondense	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no ^a
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records ^b or Number of distinct index field values ^c	Dense or Nondense	No

^aYes if every distinct value of the ordering field starts a new block; no otherwise.

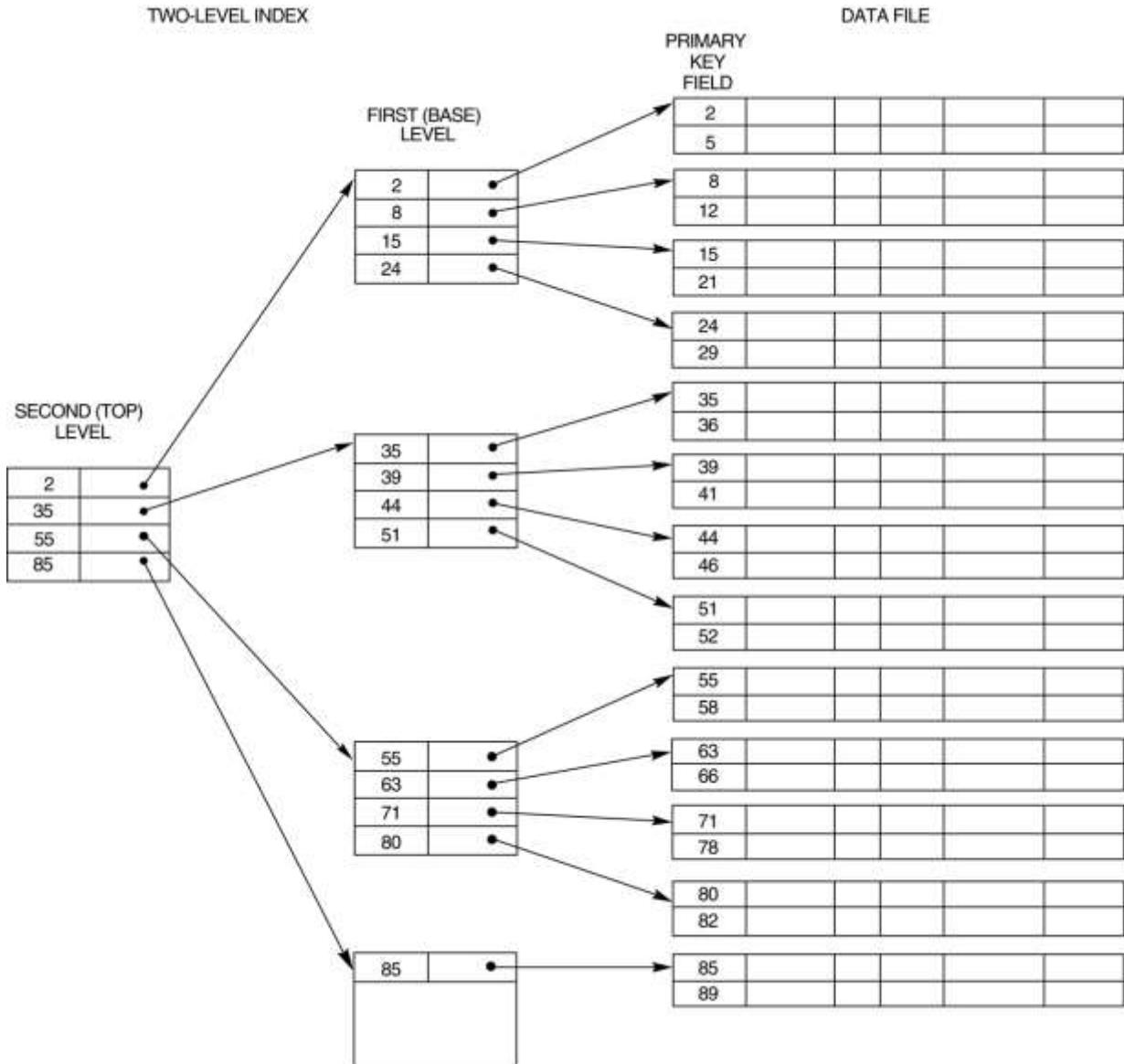
^bFor option 1.

^cFor options 2 and 3.

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*; in this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

FIGURE 14.6
 A two-level primary index resembling
 ISAM (Indexed
 Sequential
 Access Method)
 organization.



Multi-Level Indexes

- Such a multi-level index is a form of *search tree*; however, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

FIGURE 14.8
A node in a search tree with pointers to subtrees below it.

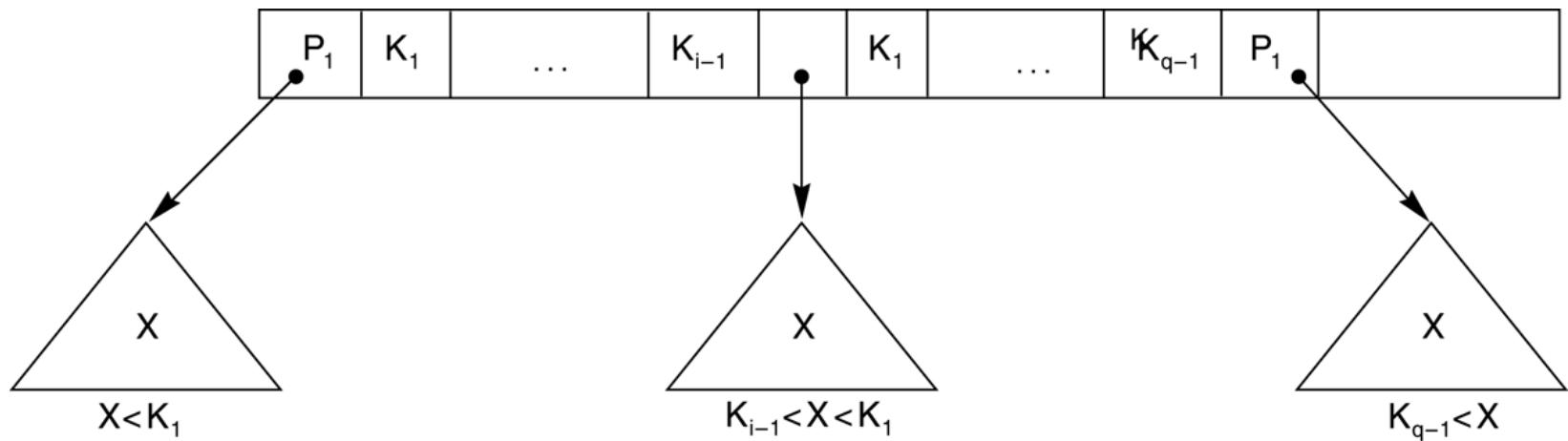
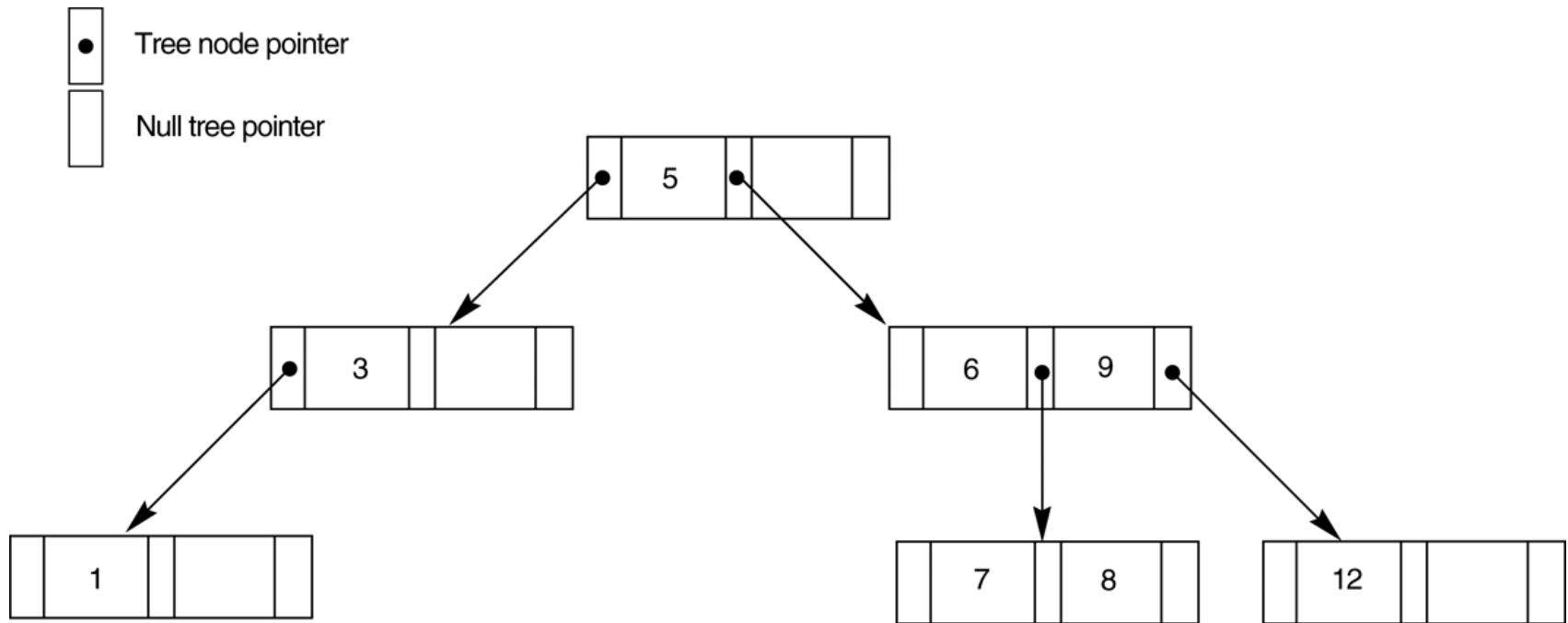


FIGURE 14.9
A search tree of order $p = 3$.



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

- Because of the insertion and deletion problem, most multi-level indexes use B-tree or B+-tree data structures, which leave space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

Dynamic Multilevel Indexes Using B-Trees and B+-Trees (contd.)

- An insertion into a node that is not full is quite efficient; if a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

FIGURE 14.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

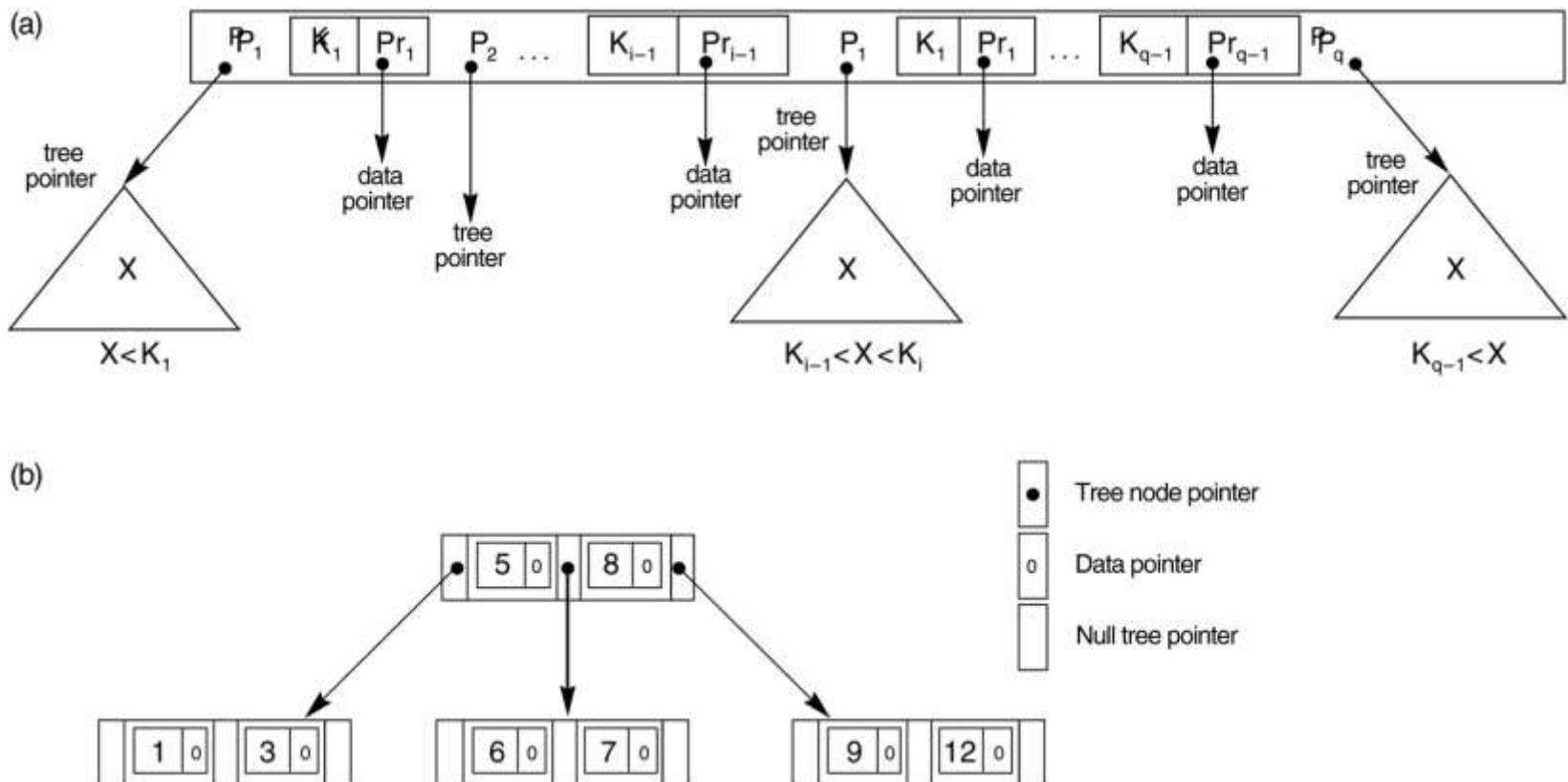


FIGURE 14.11

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

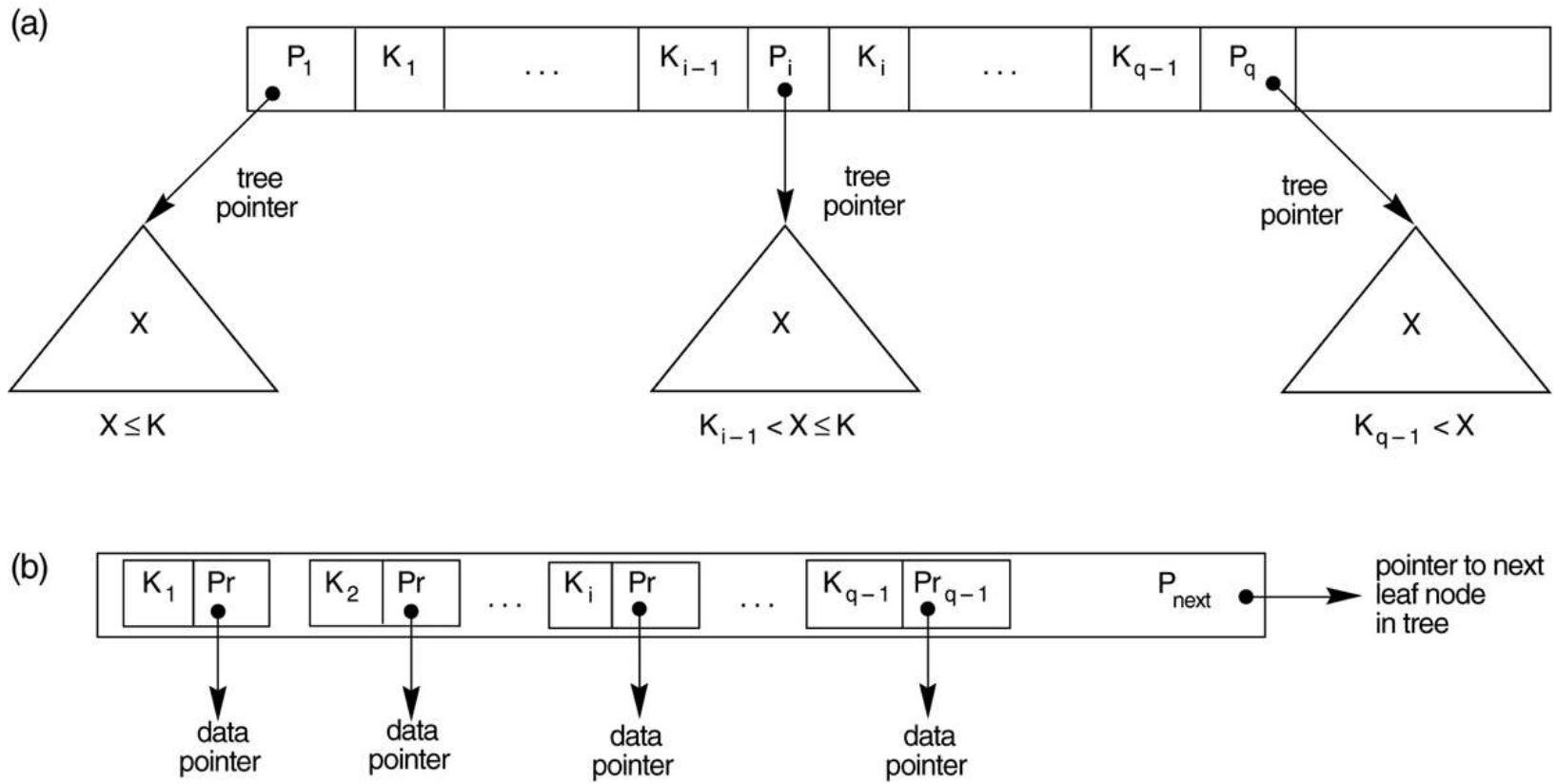


FIGURE 14.12
 An example of insertion
 in a B+-tree with $q = 3$
 and $p_{leaf} = 2$.

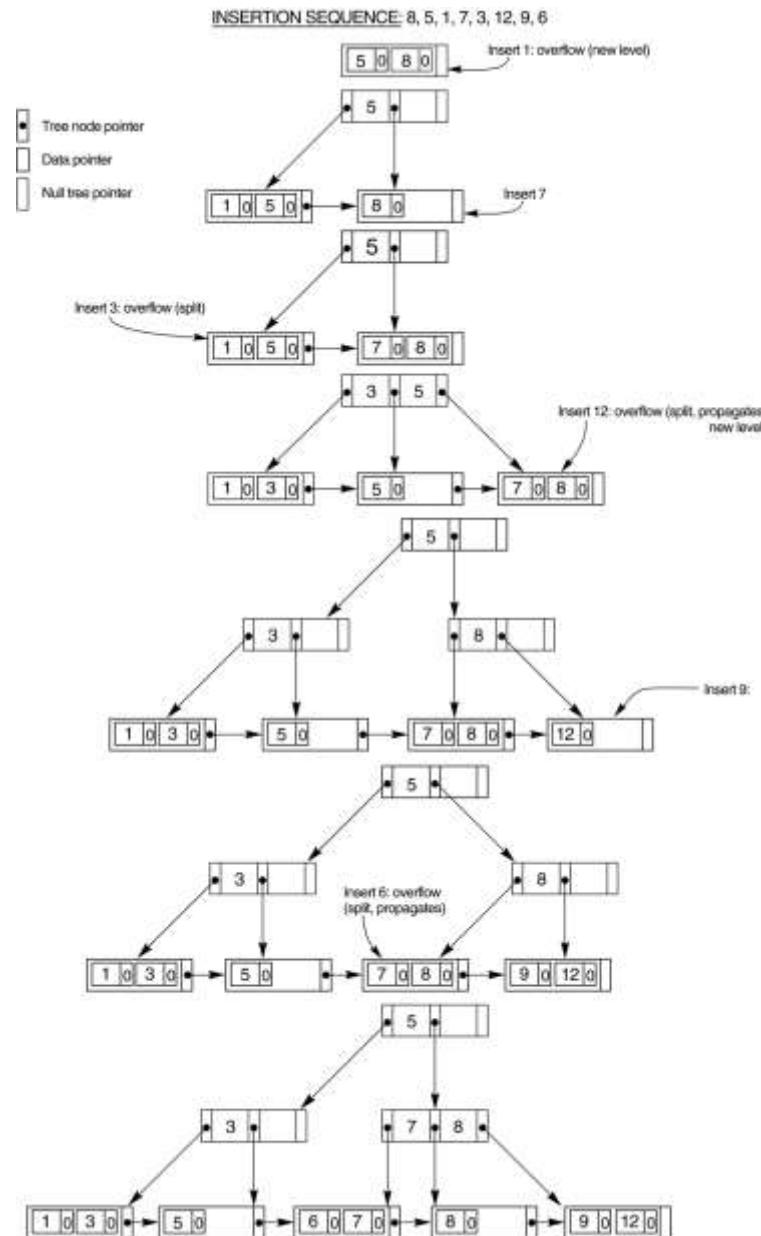
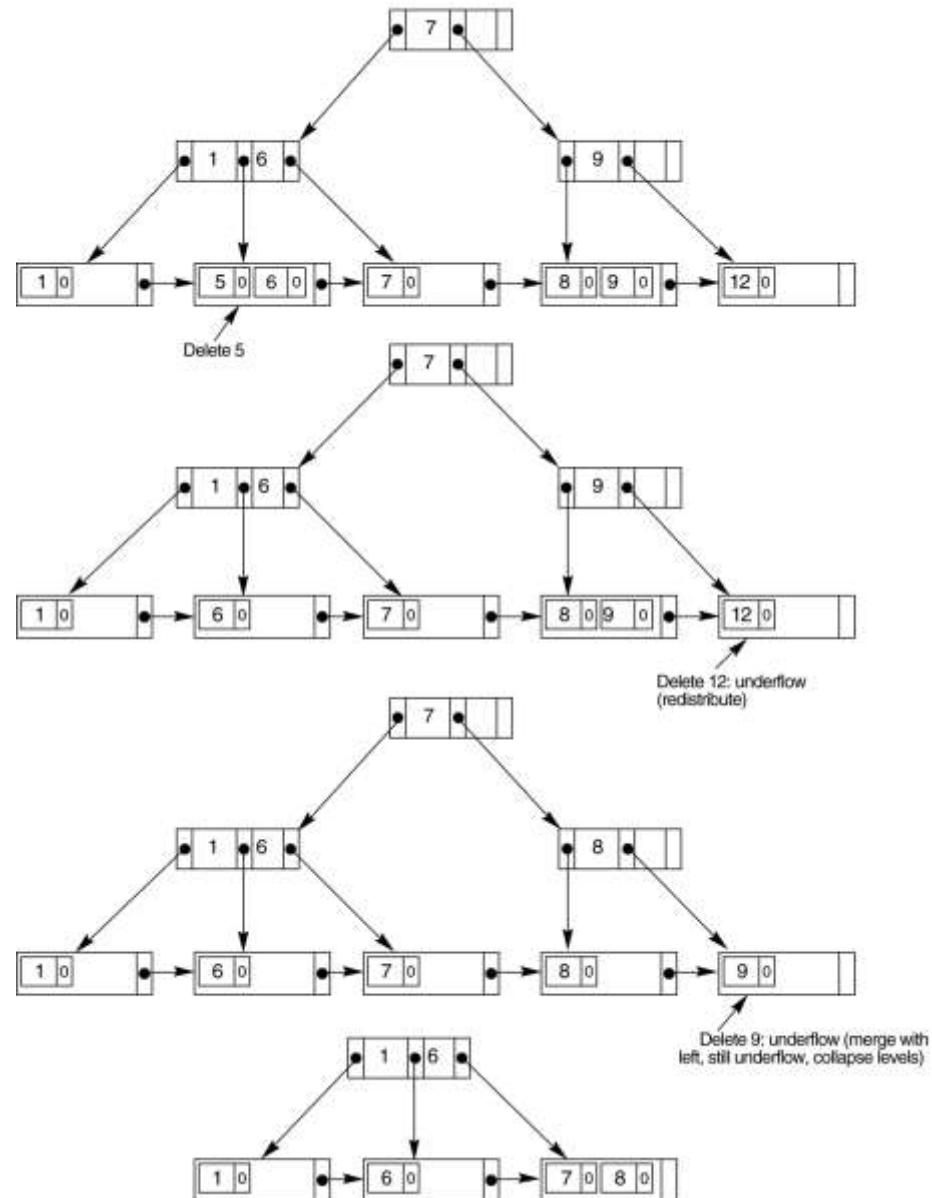


FIGURE 14.13
An example of
deletion from a
B+-tree.

DELETION SEQUENCE: 5, 12, 9



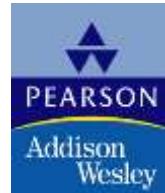
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 15

Algorithms for Query Processing and Optimization



Chapter Outline (1)

0. Introduction to Query Processing
1. Translating SQL Queries into Relational Algebra
2. Algorithms for External Sorting
3. Algorithms for SELECT and JOIN Operations
4. Algorithms for PROJECT and SET Operations
5. Implementing Aggregate Operations and Outer Joins
6. Combining Operations using Pipelining
7. Using Heuristics in Query Optimization

Chapter Outline (2)

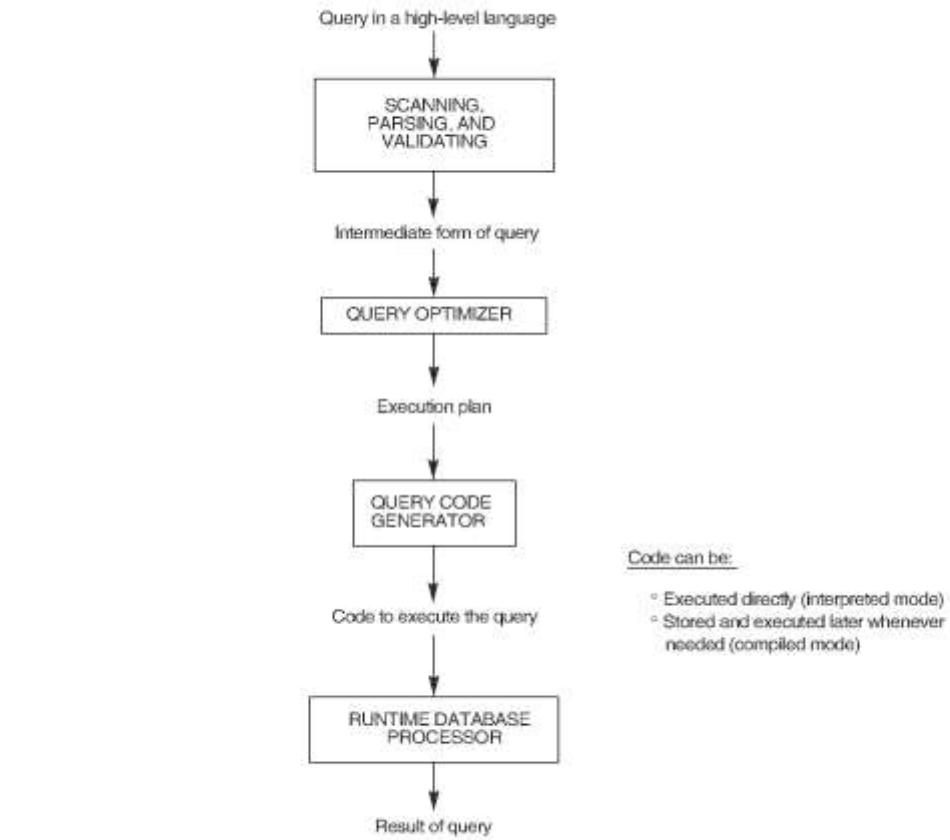
8. Using Selectivity and Cost Estimates in Query Optimization
9. Overview of Query Optimization in Oracle
10. Semantic Query Optimization

0. Introduction to Query Processing (1)

- **Query optimization:** the process of choosing a suitable execution strategy for processing a query.
- Two internal representations of a query
 - **Query Tree**
 - **Query Graph**

Introduction to Query Processing (2)

Figure 18.1 Typical steps when processing a high-level query.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

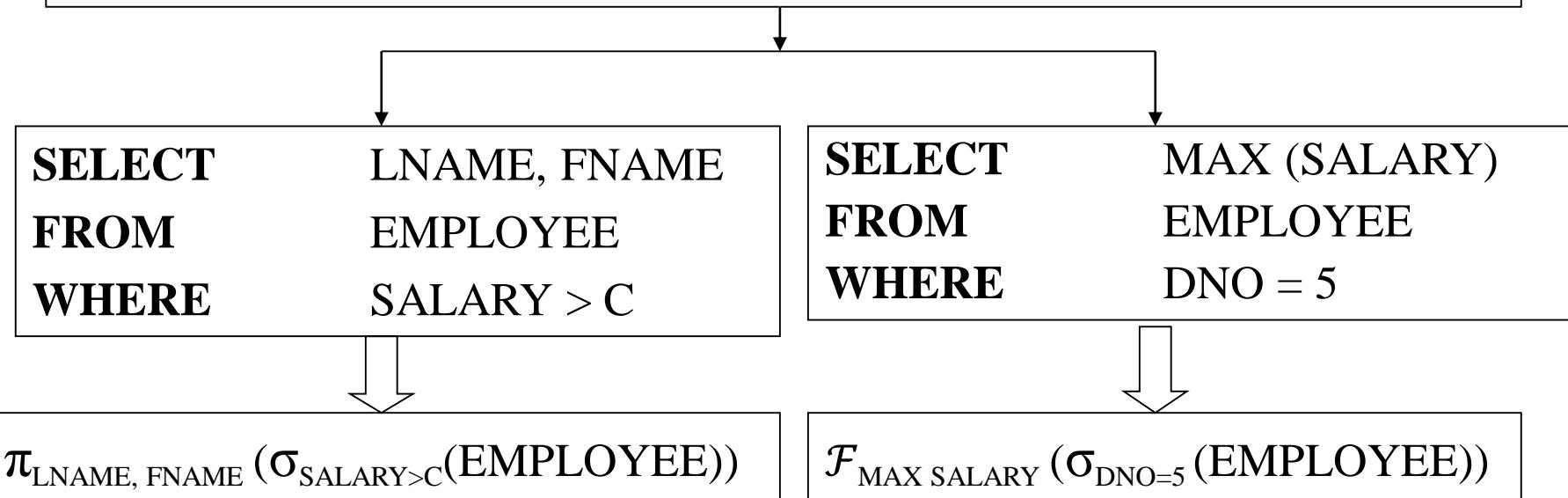
Note: The above figure is now called Figure 15.1 in Edition 4

1. Translating SQL Queries into Relational Algebra (1)

- **Query block:** the basic unit that can be translated into the algebraic operators and optimized.
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- Aggregate operators in SQL must be included in the extended algebra.

Translating SQL Queries into Relational Algebra (2)

```
SELECT          LNAME, FNAME  
FROM            EMPLOYEE  
WHERE          SALARY > (SELECT          MAX (SALARY)  
                      FROM            EMPLOYEE  
                      WHERE          DNO = 5);
```



2. Algorithms for External Sorting (1)

- **External sorting:** refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.
- **Sort-Merge strategy:** starts by sorting small subfiles (**runs**) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.
 - Sorting phase: $n_R = \lceil (b/n_B) \rceil$
 - Merging phase: $d_M = \text{Min} (n_B - 1, n_R); n_P = \lceil \log_{d_M}(n_R) \rceil$

n_R : number of initial runs; b : number of file blocks;
 n_B : available buffer space; d_M : degree of merging;
 n_P : number of passes.

Algorithms for External Sorting (2)

Figure 18.2 Outline of the sort-merge algorithm for external sorting.

```
set   i ← 1;
      j ← b; {size of the file in blocks}
      k ← nb; {size of buffer in blocks}
      m ← ⌈(j/k)⌉;
{Sort Phase}
while (i <= m)
  do {
    read next k blocks of the file into the buffer or if there are less than k blocks remaining,
       then read in the remaining blocks;
    sort the blocks in the buffer and write as a temporary subfile;
    i ← i + 1;
  }
{Merge Phase: merge subfiles until only 1 remains}
set   i ← 1;
      p ← logk-1m; {p is the number of passes for the merging phase}
      j ← m;
while (i <= p)
  do {
    n ← 1;
    q ← ⌈(j/(k-1))⌉; {number of subfiles to write in this pass}
    while (n <= q)
      do {
        read next k-1 subfiles or remaining subfiles (from previous pass) one block at a time;
        merge and write as new subfile;
        n ← n + 1;
      }
    j ← q;
    i ← i + 1;
  }
```

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.2 in Edition 4

3. Algorithms for SELECT and JOIN Operations (1)

Implementing the SELECT Operation:

- Examples:

(OP1): $\sigma_{\text{SSN}='123456789'}(\text{EMPLOYEE})$

(OP2): $\sigma_{\text{DNUMBER}>5}(\text{DEPARTMENT})$

(OP3): $\sigma_{\text{DNO}=5}(\text{EMPLOYEE})$

(OP4): $\sigma_{\text{DNO}=5 \text{ AND } \text{SALARY}>30000 \text{ AND } \text{SEX}=F}(\text{EMPLOYEE})$

(OP5): $\sigma_{\text{ESSN}=123456789 \text{ AND } \text{PNO}=10}(\text{WORKS_ON})$

Algorithms for SELECT and JOIN Operations (2)

Implementing the SELECT Operation (cont.):

Search Methods for Simple Selection:

- S1. **Linear search** (brute force): Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition.
- S2. **Binary search**: If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used. (See OP1).
- S3. **Using a primary index or hash key** to retrieve a single record: If the selection condition involves an equality comparison on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to retrieve the record.

Algorithms for SELECT and JOIN Operations (3)

Implementing the SELECT Operation (cont.):

Search Methods for Simple Selection:

- **S4. Using a primary index** to retrieve multiple records: If the comparison condition is $>$, \geq , $<$, or \leq on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
- **S5. Using a clustering index** to retrieve multiple records: If the selection condition involves an equality comparison on a non-key attribute with a clustering index, use the clustering index to retrieve all the records satisfying the selection condition.

Algorithms for SELECT and JOIN Operations (4)

Implementing the SELECT Operation (cont.):

Search Methods for Simple Selection:

- S6. **Using a secondary (B+-tree) index:** On an equality comparison, this search method can be used to retrieve a single record if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key. In addition, it can be used to retrieve records on conditions involving $>$, \geq , $<$, or \leq . (**FOR RANGE QUERIES**)

Algorithms for SELECT and JOIN Operations (5)

Implementing the SELECT Operation (cont.):

Search Methods for Complex Selection:

- S7. **Conjunctive selection:** If an attribute involved in any single *simple condition* in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
- S8. **Conjunctive selection using a composite index:** If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined field, we can use the index directly.

Algorithms for SELECT and JOIN Operations (6)

Implementing the SELECT Operation (cont.):

Search Methods for Complex Selection:

- S9. **Conjunctive selection by intersection of record pointers:** This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers). Each index can be used to retrieve the *record pointers* that satisfy the individual condition. The *intersection* of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

Algorithms for SELECT and JOIN Operations (7)

Implementing the SELECT Operation (cont.):

- Whenever a **single condition** specifies the selection, we can only check whether an access path exists on the attribute involved in that condition. If an access path exists, the method corresponding to that access path is used; otherwise, the “brute force” linear search approach of method S1 is used. (See OP1, OP2 and OP3)
- For **conjunctive selection conditions**, whenever *more than one* of the attributes involved in the conditions have an access path, query optimization should be done to choose the access path that *retrieves the fewest records* in the most efficient way .
- **Disjunctive selection conditions**

Algorithms for SELECT and JOIN Operations (8)

Implementing the JOIN Operation:

- Join (EQUIJOIN, NATURAL JOIN)
 - two-way join: a join on two files
 - e.g. $R \bowtie_{A=B} S$
 - multi-way joins: joins involving more than two files.
 - e.g. $R \bowtie_{A=B} S \bowtie_{C=D} T$

● Examples

(OP6): EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT

(OP7): DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

Algorithms for SELECT and JOIN Operations (9)

Implementing the JOIN Operation (cont.):

Methods for implementing joins:

- J1. **Nested-loop join** (brute force): For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.
- J2. **Single-loop join** (Using an access structure to retrieve the matching records): If an index (or hash key) exists for one of the two join attributes — say, B of S — retrieve each record t in R , one at a time, and then use the access structure to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.

Algorithms for SELECT and JOIN Operations (10)

Implementing the JOIN Operation (cont.):

Methods for implementing joins:

- J3. **Sort-merge join:** If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned in order of the join attributes, matching the records that have the same values for A and B. In this method, the records of each file are scanned only once each for matching with the other file—unless both A and B are non-key attributes, in which case the method needs to be modified slightly.

Algorithms for SELECT and JOIN Operations (11)

Implementing the JOIN Operation (cont.):

Methods for implementing joins:

- J4. **Hash-join:** The records of files R and S are both hashed to the *same hash file*, using the *same hashing function* on the join attributes A of R and B of S as hash keys. A single pass through the file with fewer records (say, R) hashes its records to the hash file buckets. A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R.

Algorithms for SELECT and JOIN Operations (12)

Figure 18.3 Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (a) Implementing the operation $T \leftarrow R \bowtie_{A=B} S$. (b) Implementing the operation $T \leftarrow \pi_{\langle \text{attribute list} \rangle}(R)$.

(a) sort the tuples in R on attribute A ; ("assume R has n tuples (records)")
sort the tuples in S on attribute B ; ("assume S has m tuples (records)")
set $i \leftarrow 1, j \leftarrow 1$
while ($i \leq n$) and ($j \leq m$)
do{ if $R(i)[A] > S(j)[B]$
 then set $j \leftarrow j + 1$
 elseif $R(i)[A] < S(j)[B]$
 then set $i \leftarrow i + 1$
 else { (" $R(i)[A] = S(j)[B]$, so we output a matched tuple")
 output the combined tuple $\langle R(i), S(j) \rangle$ to T
 ("output other tuples that match $R(i)$, if any")
 set $i \leftarrow i + 1$
 while ($i \leq n$) and ($R(i)[A] < S(j)[B]$)
 do { output the combined tuple $\langle R(i), S(j) \rangle$ to T
 set $i \leftarrow i + 1$
 }
 ("output other tuples that match $S(j)$, if any")
 set $k \leftarrow i + 1$
 while ($k \leq n$) and ($R(k)[A] < S(j)[B]$)
 do { output the combined tuple $\langle R(k), S(j) \rangle$ to T
 set $k \leftarrow k + 1$
 }
 set $i \leftarrow k, j \leftarrow j + 1$
 }
}

(b) create a tuple $t[\langle \text{attribute list} \rangle]$ in T' for each tuple t in R ;
 (" T' contains the projection result before duplicate elimination")
 if $\langle \text{attribute list} \rangle$ includes a key of R
 the $T' \leftarrow T'$
 else { sort the tuples in T'
 set $A \leftarrow 1, j \leftarrow 2$
 while $i \leq n$
 do { output the tuple $T'[i]$ to T
 while $T'[i] = T'[j]$ and $j \leq n$ do $j \leftarrow j + 1$; ("eliminate duplicates")
 i $\leftarrow j, j \leftarrow j + 1$
 }
 }

 (" T contains the projection result after duplicate elimination ")

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.3 in Edition 4

Algorithms for SELECT and JOIN Operations (13)

Figure 18.3 Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples.

(c) Implementing the operation $T \leftarrow R \cup S$. (d) Implementing the operation $T \leftarrow R \cap S$. (e) Implementing the operation $T \leftarrow R - S$.

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

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

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

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.3 (continued) in Edition 4

Algorithms for SELECT and JOIN Operations (14)

Implementing the JOIN Operation (cont.):

- Factors affecting JOIN performance
 - Available buffer space
 - Join selection factor
 - Choice of inner VS outer relation

Algorithms for SELECT and JOIN Operations

(15)

Implementing the JOIN Operation (cont.):

Other types of JOIN algorithms

- **Partition hash join**

- **Partitioning phase:** Each file (R and S) is first partitioned into M partitions using a partitioning hash function on the join attributes:

$R_1, R_2, R_3, \dots, R_m$ and $S_1, S_2, S_3, \dots, S_m$

Minimum number of in-memory buffers needed for the partitioning phase: $M+1$.

A disk sub-file is created per partition to store the tuples for that partition.

- **Joining or probing phase:** Involves M iterations, one per partitioned file. Iteration i involves joining partitions R_i and S_i .

Algorithms for SELECT and JOIN Operations (16)

Implementing the JOIN Operation (cont.):

Partitioned Hash Join Procedure:

Assume R_i is smaller than S_i .

1. Copy records from R_i into memory buffers.
2. Read all blocks from S_i , one at a time and each record from S_i is used to *probe* for a matching record(s) from partition S_i .
3. Write matching record from R_i after joining to the record from S_i into the result file.

Algorithms for SELECT and JOIN Operations

(17)

Implementing the JOIN Operation (cont.):

Cost analysis of partition hash join:

1. Reading and writing each record from R and S during the partitioning phase: $(b_R + b_S), (b_R + b_S)$
2. Reading each record during the joining phase: $(b_R + b_S)$
3. Writing the result of join: b_{RES}

Total Cost: $3 * (b_R + b_S) + b_{RES}$

Algorithms for SELECT and JOIN Operations

(18)

Implementing the JOIN Operation (cont.):

- **Hybrid hash join:** Same as partitioned hash join except:
Joining phase of one of the partitions is included during the partitioning phase.
 - **Partitioning phase:** Allocate buffers for smaller relation-one block for each of the $M-1$ partitions, remaining blocks to partition 1. Repeat for the larger relation in the pass through $S.$)
 - **Joining phase:** $M-1$ iterations are needed for the partitions $R_2, R_3, R_4, \dots, R_m$ and $S_2, S_3, S_4, \dots, S_m$. R_1 and S_1 are joined during the partitioning of S_1 , and results of joining R_1 and S_1 are already written to the disk by the end of partitioning phase .

4. Algorithms for PROJECT and SET Operations (1)

Algorithm for PROJECT operations (Figure 15.3b)

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

1. If $\langle \text{attribute list} \rangle$ has a key of relation R, extract all tuples from R with only the values for the attributes in $\langle \text{attribute list} \rangle$.
2. If $\langle \text{attribute list} \rangle$ does NOT include a key of relation R, duplicated tuples must be removed from the results.

- **Methods to remove duplicate tuples**
 1. **Sorting**
 2. **Hashing**

Algorithms for PROJECT and SET Operations (2)

Algorithm for SET operations

- **Set operations:** UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT.
- **CARTESIAN PRODUCT** of relations R and S include all possible combinations of records from R and S. The attribute of the result include all attributes of R and S.
- **Cost analysis** of CARTESIAN PRODUCT
If R has n records and j attributes and S has m records and k attributes, the result relation will have $n*m$ records and $j+k$ attributes.
- CARTESIAN PRODUCT operation is very **expensive** and should be avoided if possible.

Algorithms for PROJECT and SET Operations (3)

Algorithm for SET operations (Cont.)

- **UNION** (See Figure 15.3c)
 1. Sort the two relations on the same attributes.
 2. Scan and merge both sorted files concurrently, whenever the same tuple exists in both relations, only one is kept in the merged results.
- **INTERSECTION** (See Figure 15.3d)
 1. Sort the two relations on the same attributes.
 2. Scan and merge both sorted files concurrently, keep in the merged results only those tuples that appear in both relations.
- **SET DIFFERENCE R-S** (See Figure 15.3e)
(keep in the merged results only those tuples that appear in relation R but not in relation S.)

5. Implementing Aggregate Operations and Outer Joins (1)

Implementing Aggregate Operations:

- Aggregate operators: MIN, MAX, SUM, COUNT and AVG
- Options to implement aggregate operators:
 - Table Scan
 - Index
- Example

```
SELECT      MAX (SALARY)  
FROM        EMPLOYEE;
```

If an (ascending) index on SALARY exists for the employee relation, then the optimizer could decide on traversing the index for the largest value, which would entail following the right most pointer in each index node from the root to a leaf.

Implementing Aggregate Operations and Outer Joins (2)

Implementing Aggregate Operations (cont.):

- **SUM, COUNT and AVG**
- 1. For a **dense index** (each record has one index entry): apply the associated computation to the values in the index.
- 2. For a **non-dense index**: actual number of records associated with each index entry must be accounted for
- With **GROUP BY**: the aggregate operator must be applied separately to each group of tuples.
 1. Use sorting or hashing on the group attributes to partition the file into the appropriate groups;
 2. Computes the aggregate function for the tuples in each group.
- What if we have **Clustering index** on the grouping attribute?

Implementing Aggregate Operations and Outer Joins (3)

Implementing Outer Join:

- **Outer Join Operators:** LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL OUTER JOIN.
- The full outer join produces a result which is equivalent to the union of the results of the left and right outer joins.
- **Example**

```
SELECT          FNAME, DNAME  
FROM           (EMPLOYEE LEFT OUTER JOIN DEPARTMENT  
    ON DNO = DNUMBER);
```

Note: The result of this query is a table of employee names and their associated departments. It is similar to a regular join result, with the exception that if an employee does not have an associated department, the employee's name will still appear in the resulting table, although the department name would be indicated as null.

Implementing Aggregate Operations and Outer Joins (4)

Implementing Outer Join (cont.):

- **Modifying Join Algorithms:** Nested Loop or Sort-Merge joins can be modified to implement outer join. e.g., for left outer join, use the left relation as outer relation and construct result from every tuple in the left relation. If there is a match, the concatenated tuple is saved in the result. However, if an outer tuple does not match, then the tuple is still included in the result but is padded with a null value(s).

Implementing Aggregate Operations and Outer Joins (5)

Implementing Outer Join (cont.):

- Executing a combination of relational algebra operators.

Implement the previous left outer join example

1. {Compute the JOIN of the EMPLOYEE and DEPARTMENT tables}
$$\text{TEMP1} \leftarrow \pi_{\text{FNAME}, \text{DNAME}}(\text{EMPLOYEE} \bowtie \text{DEPARTMENT} \text{ DNO=DNUMBER})$$
 2. {Find the EMPLOYEES that do not appear in the JOIN}
$$\text{TEMP2} \leftarrow \pi_{\text{FNAME}}(\text{EMPLOYEE}) - \pi_{\text{FNAME}}(\text{Temp1})$$
 3. {Pad each tuple in TEMP2 with a null DNAME field}
$$\text{TEMP2} \leftarrow \text{TEMP2} \times \text{'null'}$$
 4. {UNION the temporary tables to produce the LEFT OUTER JOIN result}
$$\text{RESULT} \leftarrow \text{TEMP1} \cup \text{TEMP2}$$
- The cost of the outer join, as computed above, would include the cost of the associated steps (i.e., join, projections and union).

6. Combining Operations using Pipelining (1)

● Motivation

- A query is mapped into a sequence of operations.
- Each execution of an operation produces a temporary result.
- Generating and saving temporary files on disk is time consuming and expensive.

● Alternative:

- Avoid constructing temporary results as much as possible.
- Pipeline the data through multiple operations - pass the result of a previous operator to the next without waiting to complete the previous operation.

Combining Operations using Pipelining (2)

- **Example:** For a 2-way join, combine the 2 selections on the input and one projection on the output with the Join.
- Dynamic generation of code to allow for multiple operations to be pipelined.
- Results of a select operation are fed in a "**Pipeline**" to the join algorithm.
- Also known as **stream-based processing**.

7. Using Heuristics in Query Optimization(1)

- **Process for heuristics optimization**
 1. The parser of a high-level query generates an *initial internal representation*;
 2. Apply heuristics rules to optimize the internal representation.
 3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The **main heuristic** is to apply first the operations that reduce the size of intermediate results.
E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Using Heuristics in Query Optimization (2)

- **Query tree:** 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 are available and then replacing that internal node by the relation that results from executing the operation.
- **Query graph:** a graph data structure that corresponds to a relational calculus expression. It does **not** indicate an order on which operations to perform first. There is only a **single** graph corresponding to each query.

Using Heuristics in Query Optimization (3)

- **Example:**

For every project located in ‘Stafford’, retrieve the project number, the controlling department number and the department manager’s last name, address and birthdate.

Relation algebra:

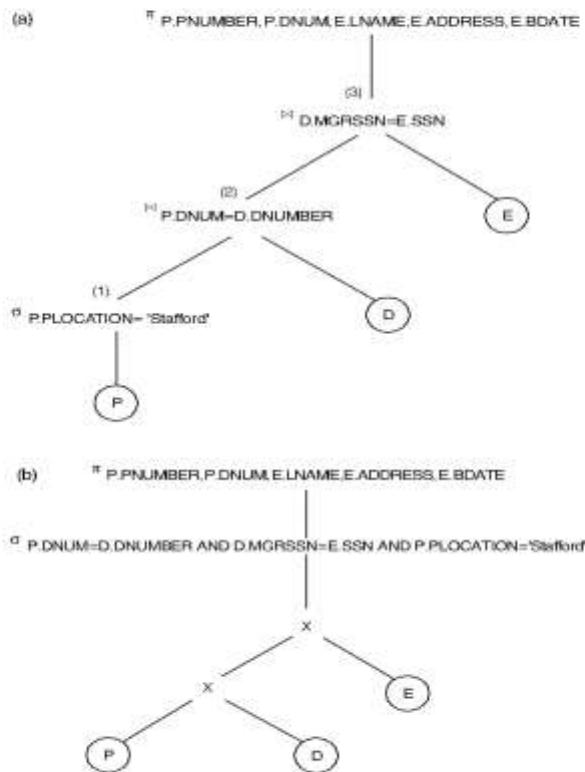
$$\pi_{PNUMBER, DNUM, LNAME, ADDRESS, BDATE} (((\sigma_{PLOCATION='STAFFORD'}(PROJECT)) \bowtie_{DNUM=DNUMBER} (DEPARTMENT)) \bowtie_{MGRSSN=SSN} (EMPLOYEE))$$

SQL query:

Q2: SELECT P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
P.PLOCATION='STAFFORD';

Using Heuristics in Query Optimization (4)

Figure 18.4 Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2.

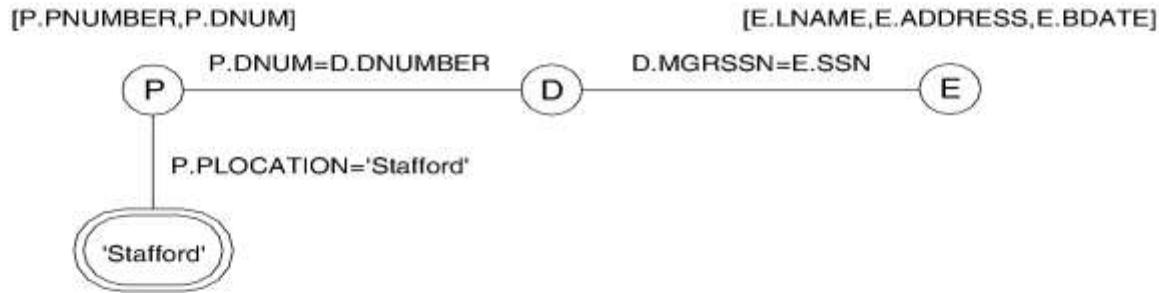


© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.4 in Edition 4

Using Heuristics in Query Optimization (5)

Figure 18.4 (c) Query graph for Q2.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.4 (continued) in Edition 4

Using Heuristics in Query Optimization (6)

Heuristic Optimization of Query Trees:

- The same query could correspond to many different relational algebra expressions — and hence many different query trees.
- The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.
- **Example:**

Q: SELECT LNAME

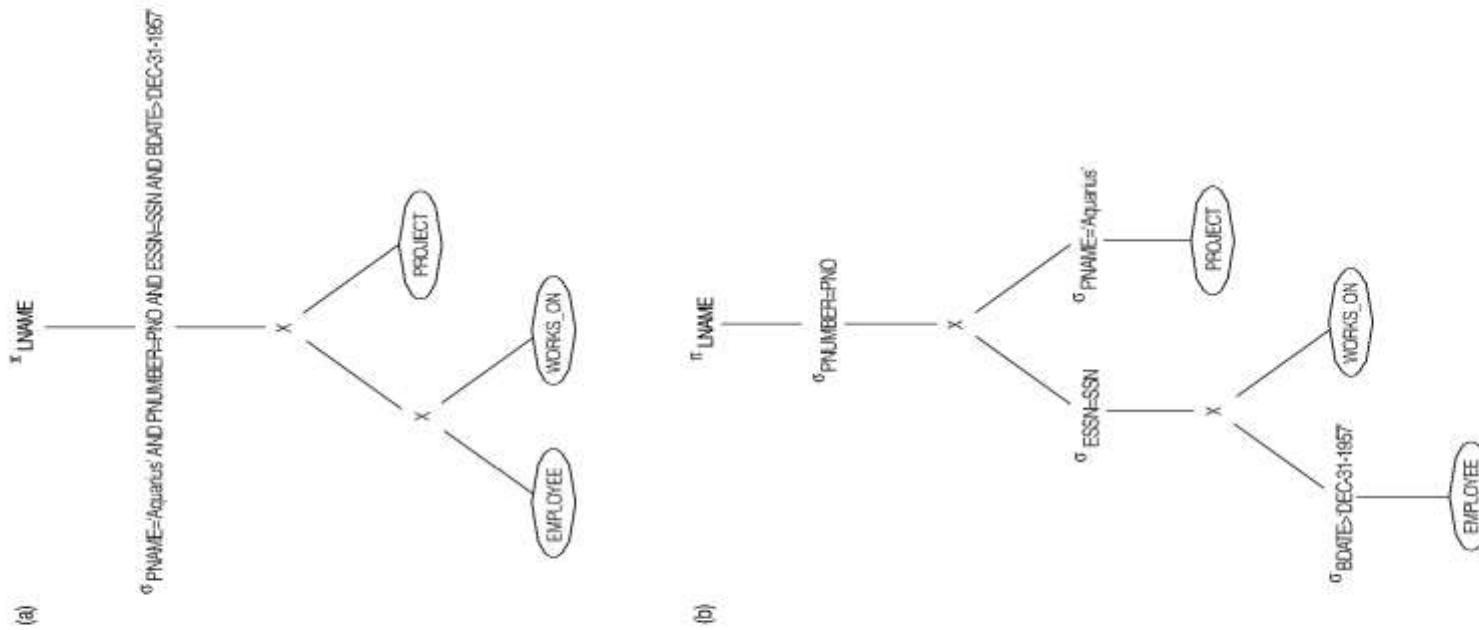
FROM EMPLOYEE, WORKS_ON, PROJECT

WHERE PNAME = ‘AQUARIUS’ AND PNMUBER=PNO

AND ESSN=SSN AND BDATE > ‘1957-12-31’;

Using Heuristics in Query Optimization (7)

Figure 18.5 Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree.

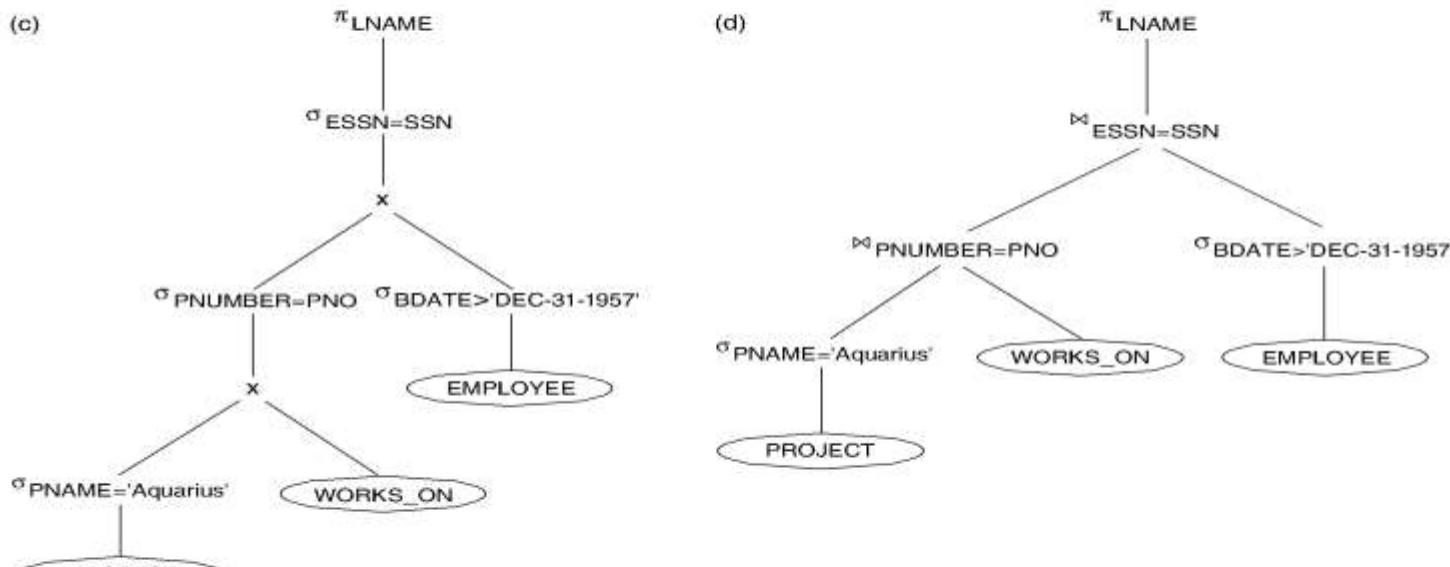


© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.5 in Edition 4

Using Heuristics in Query Optimization (8)

Figure 18.5 Steps in converting a query tree during heuristic optimization. (c) Applying the more restrictive SELECT operation first. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

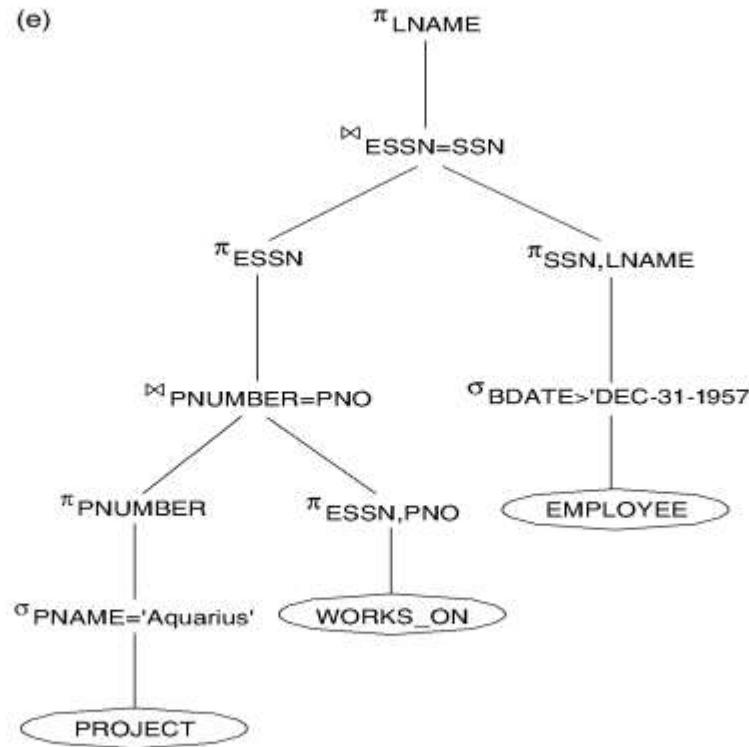


© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.5(continued c, d) in Edition 4

Using Heuristics in Query Optimization (9)

Figure 18.5 Steps in converting a query tree during heuristic optimization. (e) Moving PROJECT operations down the query tree.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 15.5(continued e) in Edition 4

Using Heuristics in Query Optimization (10)

General Transformation Rules for Relational Algebra Operations:

1. Cascade of σ : A conjunctive selection condition can be broken up into a cascade (sequence) of individual σ operations:
$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) = \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$
2. Commutativity of σ : The σ operation is commutative:
$$\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$$
3. Cascade of π : In a cascade (sequence) of π operations, all but the last one can be ignored:
$$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\dots(\pi_{\text{List}_n}(R))\dots)) = \pi_{\text{List}_1}(R)$$
4. Commuting σ with π : If the selection condition c involves only the attributes A_1, \dots, A_n in the projection list, the two operations can be commuted:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) = \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

Elmasri and Navathe, Fundamentals of Database Systems, Fourth Edition

Using Heuristics in Query Optimization (11)

General Transformation Rules for Relational Algebra Operations (cont.):

5. Commutativity of \bowtie (and x): The \bowtie operation is commutative as is the x operation $\bowtie R \underset{c}{\bowtie} S = S \underset{c}{\bowtie} R; R x S = S x R$ \bowtie
6. Commuting σ with \bowtie (or x): If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R —the two operations can be commuted as follows :

$$\sigma_c (R \bowtie S) = (\sigma_c (R)) \bowtie S$$

Alternatively, if the selection condition c can be written as $(c_1 \text{ and } c_2)$, where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S , the operations commute as follows:

$$\sigma_c (R \bowtie S) = (\sigma_{c_1} (R)) \bowtie (\sigma_{c_2} (S))$$

Using Heuristics in Query Optimization (12)

General Transformation Rules for Relational Algebra Operations (cont.):

7. Commuting π with \bowtie (or x): Suppose that the projection list is $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, where A_1, \dots, A_n are attributes of R and B_1, \dots, B_m are attributes of S . If the join condition c involves only attributes in L , the two operations can be commuted as follows:

$$\pi_L (R \bowtie_c S) = (\pi_{A_1, \dots, A_n} (R) \bowtie_c (\pi_{B_1, \dots, B_m} (S))$$

If the join condition c contains additional attributes not in L , these must be added to the projection list, and a final π operation is needed.

Using Heuristics in Query Optimization (13)

General Transformation Rules for Relational Algebra Operations (cont.):

8. Commutativity of set operations: The set operations \cup and \cap are commutative but $-$ is not.
9. Associativity of \bowtie , \times , \cup , and \cap : These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have
$$(R \theta S) \theta T = R \theta (S \theta T)$$
10. Commuting σ with set operations: The σ operation commutes with \cup , \cap , and $-$. If θ stands for any one of these three operations, we have

$$\sigma_c(R \theta S) = (\sigma_c(R)) \theta (\sigma_c(S))$$

Using Heuristics in Query Optimization (14)

General Transformation Rules for Relational Algebra Operations (cont.):

11. The π operation commutes with ν .

$$\pi_L(R \nu S) = (\pi_L(R)) \nu (\pi_L(S))$$

12. Converting a (σ, x) sequence into \bowtie : If the condition c of a σ that follows a x corresponds to a join condition, convert the (σ, x) sequence into a \bowtie as follows:

$$(\sigma_C(R x S)) = (R \bowtie_C S)$$

13. Other transformations

Using Heuristics in Query Optimization (15)

Outline of a Heuristic Algebraic Optimization Algorithm:

1. Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.
2. Using rules 2, 4, 6, and 10 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.
3. Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
4. Using Rule 12, combine a cartesian product operation with a subsequent select operation in the tree into a join

Using Heuristics in Query Optimization (16)

Outline of a Heuristic Algebraic Optimization Algorithm (cont.)

5. Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.

6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Using Heuristics in Query Optimization (17)

Summary of Heuristics for Algebraic Optimization:

1. The main heuristic is to apply first the operations that reduce the size of intermediate results.
2. Perform select operations as early as possible to reduce the number of tuples and perform project operations as early as possible to reduce the number of attributes.
(This is done by moving select and project operations as far down the tree as possible.)
3. The select and join operations that are most restrictive should be executed before other similar operations. (This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.)

Using Heuristics in Query Optimization (17)

Query Execution Plans

- An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
- **Materialized evaluation:** the result of an operation is stored as a temporary relation.
- **Pipelined evaluation:** as the result of an operator is produced, it is forwarded to the next operator in sequence.

8. Using Selectivity and Cost Estimates in Query Optimization (1)

- **Cost-based query optimization:** Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
(Compare to heuristic query optimization)
- **Issues**
 - **Cost function**
 - **Number of execution strategies to be considered**

Using Selectivity and Cost Estimates in Query Optimization (2)

- **Cost Components for Query Execution**
 1. Access cost to secondary storage
 2. Storage cost
 3. Computation cost
 4. Memory usage cost
 5. Communication cost

Note: Different database systems may focus on different cost components.

Using Selectivity and Cost Estimates in Query Optimization (3)

- **Catalog Information Used in Cost Functions**
 - Information about the size of a file
 - **number of records (tuples) (r),**
 - **record size (R),**
 - **number of blocks (b)**
 - **blocking factor (bfr)**
 - Information about indexes and indexing attributes of a file
 - **Number of levels (x) of each multilevel index**
 - **Number of first-level index blocks (b_{I1})**
 - **Number of distinct values (d) of an attribute**
 - **Selectivity (sl) of an attribute**
 - **Selection cardinality (s) of an attribute. ($s = sl * r$)**

Using Selectivity and Cost Estimates in Query Optimization (4)

Examples of Cost Functions for SELECT

- **S1. Linear search (brute force) approach**

$$C_{S1a} = b;$$

For an equality condition on a key, $C_{S1a} = (b/2)$ if the record is found; otherwise $C_{S1a} = b$.

- **S2. Binary search:**

$$C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$$

For an equality condition on a unique (key) attribute,

$$C_{S2} = \log_2 b$$

- **S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record**

$C_{S3a} = x + 1$; $C_{S3b} = 1$ for static or linear hashing;

$C_{S3b} = 1$ for extendible hashing;

Using Selectivity and Cost Estimates in Query Optimization (5)

Examples of Cost Functions for SELECT (cont.)

- **S4. Using an ordering index to retrieve multiple records:**
For the comparison condition on a key field with an ordering index, $C_{S4} = x + (b/2)$
- **S5. Using a clustering index to retrieve multiple records:**
 $C_{S5} = x + \lceil (s/bfr) \rceil$
- **S6. Using a secondary (B^+ -tree) index:**
For an equality comparison, $C_{S6a} = x + s$;
For an comparison condition such as $>$, $<$, \geq , or \leq ,
 $C_{S6a} = x + (b_{I1}/2) + (r/2)$

Using Selectivity and Cost Estimates in Query Optimization (6)

Examples of Cost Functions for SELECT (cont.)

- **S7. Conjunctive selection:**

Use either S1 or one of the methods S2 to S6 to solve.

For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.

- **S8. Conjunctive selection using a composite index:**

Same as S3a, S5 or S6a, depending on the type of index.

- **Examples of using the cost functions.**

Using Selectivity and Cost Estimates in Query Optimization (7)

Examples of Cost Functions for JOIN

- **Join selectivity (js)**

$$js = |(R \bowtie_C S)| / |R \times S| = |(R \bowtie_C S)| / (|R| * |S|)$$

If condition C does not exist, $js = 1$;

If no tuples from the relations satisfy condition C, $js = 0$;

Usually, $0 \leq js \leq 1$;

- **Size of the result file after join operation**

$$|(R \bowtie_C S)| = js * |R| * |S|$$

Using Selectivity and Cost Estimates in Query Optimization (8)

Examples of Cost Functions for JOIN (cont.)

- **J1. Nested-loop join:**

$$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|) / bfr_{RS})$$

(Use R for outer loop)

- **J2. Single-loop join** (using an access structure to retrieve the matching record(s))

If an index exists for the join attribute B of S with index levels x_B , we can retrieve each record s in R and then use the index to retrieve all the matching records t from S that satisfy $t[B] = s[A]$.

The cost depends on the type of index.

Using Selectivity and Cost Estimates in Query Optimization (9)

Examples of Cost Functions for JOIN (cont.)

- **J2. Single-loop join (cont.)**

For a secondary index,

$$C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|) / bfr_{RS});$$

For a clustering index,

$$C_{J2b} = b_R + (|R| * (x_B + (s_B / bfr_B))) + ((js * |R| * |S|) / bfr_{RS});$$

For a primary index,

$$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|) / bfr_{RS});$$

If a hash key exists for one of the two join attributes — B of S

$$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|) / bfr_{RS});$$

- **J3. Sort-merge join:**

$$C_{J3a} = C_S + b_R + b_S + ((js * |R| * |S|) / bfr_{RS}); \quad (C_S: \text{Cost for sorting files})$$

Using Selectivity and Cost Estimates in Query Optimization (10)

Multiple Relation Queries and Join Ordering

- A query joining n relations will have $n-1$ join operations, and hence can have a large number of different join orders when we apply the algebraic transformation rules.
- Current query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees.
- **Left-deep tree:** a binary tree where the right child of each non-leaf node is always a base relation.
 - Amenable to pipelining
 - Could utilize any access paths on the base relation (the right child) when executing the join.

9. Overview of Query Optimization in Oracle

Oracle DBMS V8

- **Rule-based query optimization:** the optimizer chooses execution plans based on heuristically ranked operations.
(Currently it is being phased out)
- **Cost-based query optimization:** the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with lowest estimate cost. The query cost is calculated based on the estimated usage of resources such as I/O, CPU and memory needed.
- Application developers could specify **hints** to the ORACLE query optimizer. The idea is that an application developer might know more information about the data.

10. Semantic Query Optimization

- **Semantic Query Optimization:** Uses constraints specified on the database schema in order to modify one query into another query that is more efficient to execute.
- Consider the following SQL query,

```
SELECT      E.LNAME, M.LNAME  
FROM        EMPLOYEE E M  
WHERE       E.SUPERSSN=M.SSN AND E.SALARY>M.SALARY
```

Explanation: Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it need not execute the query at all because it knows that the result of the query will be empty. Techniques known as **theorem proving** can be used for this purpose.

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 16

Practical Database Design and Tuning



Chapter Outline

1. Physical Database Design in Relational Databases
2. An Overview of Database Tuning in Relational Systems.

1. Physical Database Design in Relational Databases(1)

Factors that Influence Physical Database Design

A. Analyzing the database queries and transactions

For each query, the following information is needed.

- The files that will be accessed by the query;
- The attributes on which any selection conditions for the query are specified;
- The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified;
- The attributes whose values will be retrieved by the query.

Note: the attributes listed in items 2 and 3 above are candidates for definition of access structures.

Physical Database Design in Relational Databases(2)

Factors that Influence Physical Database Design (cont.)

A. Analyzing the database queries and transactions (cont.)

For each **update** transaction or operation, the following information is needed.

- The files that will be updated;
- The type of operation on each file (insert, update or delete);
- The attributes on which selection conditions for a delete or update operation are specified;
- The attributes whose values will be changed by an update operation.

Note: the attributes listed in items 3 above are candidates for definition of access structures. However, the attributes listed in item 4 are candidates for avoiding an access structure.

Physical Database Design in Relational Databases(3)

Factors that Influence Physical Database Design (cont.)

B. Analyzing the expected frequency of invocation of queries and transactions

- The expected frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of expected frequency of use for all the queries and transactions.
- It is expressed as the expected frequency of using each attribute in each file as a selection attribute or join attribute, over all the queries and transactions.
- **80-20 rule**

Physical Database Design in Relational Databases(4)

Factors that Influence Physical Database Design (cont.)

C. Analyzing the time constraints of queries and transactions

- Performance constraints place further priorities on the attributes that are candidates for access paths.
- The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structure.

D. Analyzing the expected frequencies of update operations

A minimum number of access paths should be specified for a file that is updated frequently.

E. Analyzing the uniqueness constraints on attributes.

Access paths should be specified on all candidate key attributes — or set of attributes — that are either the primary key or constrained to be unique.

Physical Database Design in Relational Databases(5)

Physical Database Design Decisions

- **Design decisions about indexing**

1. Whether to index an attribute?
2. What attribute or attributes to index on?
3. Whether to set up a clustered index?
4. Whether to use a hash index over a tree index?
5. Whether to use dynamic hashing for the file?

Physical Database Design in Relational Databases(6)

Physical Database Design Decisions (cont.)

- Denormalization as a design decision for speeding up queries
 - The goal of normalization is to separate the logically related attributes into tables to minimize redundancy and thereby avoid the update anomalies that cause an extra processing overhead to maintain consistency of the database.
 - The goal of denormalization is to improve the performance of frequently occurring queries and transactions. (Typically the designer adds to a table attributes that are needed for answering queries or producing reports so that a join with another table is avoided.)
 - Trade off between update and query performance

2. An Overview of Database Tuning in Relational Systems (1)

- **Tuning:** the process of continuing to revise/adjust the physical database design by monitoring resource utilization as well as internal DBMS processing to reveal bottlenecks such as contention for the same data or devices.
- **Goal:**
 - To make application run faster
 - To lower the response time of queries/transactions
 - To improve the overall throughput of transactions

An Overview of Database Tuning in Relational Systems (2)

Statistics internally collected in DBMSs:

- Size of individual tables
- Number of distinct values in a column
- The number of times a particular query or transaction is submitted/executed in an interval of time
- The times required for different phases of query and transaction processing

Statistics obtained from monitoring:

- Storage statistics
- I/O and device performance statistics
- Query/transaction processing statistics
- Locking/logging related statistics
- Index statistics

An Overview of Database Tuning in Relational Systems (3)

Problems to be considered in tuning:

- How to avoid excessive lock contention?
- How to minimize overhead of logging and unnecessary dumping of data?
- How to optimize buffer size and scheduling of processes?
- How to allocate resources such as disks, RAM and processes for most efficient utilization?

An Overview of Database Tuning in Relational Systems (4)

Tuning Indexes

- **Reasons to tuning indexes**
 - Certain queries may take too long to run for lack of an index;
 - Certain indexes may not get utilized at all;
 - Certain indexes may be causing excessive overhead because the index is on an attribute that undergoes frequent changes

- **Options to tuning indexes**
 - Drop or/and build new indexes
 - Change a non-clustered index to a clustered index (and vice versa)
 - Rebuilding the index

An Overview of Database Tuning in Relational Systems (5)

Tuning the Database Design

- **Dynamically changed processing requirements** need to be addressed by making changes to the conceptual schema if necessary and to reflect those changes into the logical schema and physical design.

An Overview of Database Tuning in Relational Systems (6)

Tuning the Database Design (cont.)

- **Possible changes to the database design**

- Existing tables may be joined (denormalized) because certain attributes from two or more tables are frequently needed together.
- For the given set of tables, there may be alternative design choices, all of which achieve 3NF or BCNF. One may be replaced by the other.
- A relation of the form $R(\underline{K}, A, B, C, D, \dots)$ that is in BCNF can be stored into multiple tables that are also in BCNF by replicating the key K in each table.
- Attribute(s) from one table may be repeated in another even though this creates redundancy and potential anomalies.
- Apply **horizontal partitioning** as well as **vertical partitioning** if necessary.

An Overview of Database Tuning in Relational Systems (7)

Tuning Queries

- **Indications for tuning queries**
 - A query issues too many disk accesses
 - The query plan shows that relevant indexes are not being used.
- **Typical instances for query tuning**
 1. Many query optimizers do not use indexes in the presence of arithmetic expressions, numerical comparisons of attributes of different sizes and precision, NULL comparisons, and sub-string comparisons.
 2. Indexes are often not used for nested queries using IN;

An Overview of Database Tuning in Relational Systems (8)

Tuning Queries (cont.)

- Typical instances for query tuning (cont.)
 3. Some *DISTINCTs* may be redundant and can be avoided without changing the result.
 4. Unnecessary use of temporary result tables can be avoided by collapsing multiple queries into a single query unless the temporary relation is needed for some intermediate processing.
 5. In some situations involving using of correlated queries, temporaries are useful.
 6. If multiple options for join condition are possible, choose one that uses a clustering index and avoid those that contain string comparisons.

An Overview of Database Tuning in Relational Systems (9)

Tuning Queries (cont.)

- **Typical instances for query tuning (cont.)**
 7. The order of tables in the *FROM* clause may affect the join processing.
 8. Some query optimizers perform worse on nested queries compared to their equivalent un-nested counterparts.
 9. Many applications are based on views that define the data of interest to those applications. Sometimes these views become an overkill.

An Overview of Database Tuning in Relational Systems (10)

Additional Query Tuning Guidelines

1. A query with multiple selection conditions that are connected via *OR* may not be prompting the query optimizer to use any index. Such a query may be split up and expressed as a union of queries, each with a condition on an attribute that causes an index to be used.
2. Apply the following transformations
 - *NOT* condition may be transformed into a positive expression.
 - Embedded *SELECT* blocks may be replaced by joins.
 - If an equality join is set up between two tables, the range predicate on the joining attribute set up in one table may be repeated for the other table
3. *WHERE* conditions may be rewritten to utilize the indexes on multiple columns.

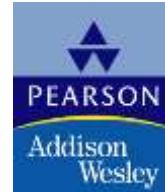
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 17

Introduction to Transaction Processing Concepts and Theory



Chapter Outline

- 1 Introduction to Transaction Processing
- 2 Transaction and System Concepts
- 3 Desirable Properties of Transactions
- 4 Characterizing Schedules based on Recoverability
- 5 Characterizing Schedules based on Serializability
- 6 Transaction Support in SQL

1 Introduction to Transaction Processing (1)

- **Single-User System:** At most one user at a time can use the system.
- **Multiuser System:** Many users can access the system concurrently.
- **Concurrency**
 - **Interleaved processing:** concurrent execution of processes is interleaved in a single CPU
 - **Parallel processing:** processes are concurrently executed in multiple CPUs.

Introduction to Transaction Processing (2)

- **A Transaction:** logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
- **A transaction (set of operations)** may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries:** Begin and End transaction.
- **An application program** may contain several transactions separated by the Begin and End transaction boundaries.

Introduction to Transaction Processing (3)

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database** - collection of named data items
- **Granularity of data** - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
 - **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named X*.
 - **write_item(X)**: Writes the value of program variable X into the database item named X.

Introduction to Transaction Processing (4)

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.
- **read_item(X) command includes the following steps:**
 1. Find the address of the disk block that contains item X.
 2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
 3. Copy item X from the buffer to the program variable named X.

Introduction to Transaction Processing (5)

READ AND WRITE OPERATIONS (cont.):

- **write_item(X) command includes the following steps:**
 1. Find the address of the disk block that contains item X.
 2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
 3. Copy item X from the program variable named X into its correct location in the buffer.
 4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

FIGURE 17.2

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);
```

Introduction to Transaction Processing (7)

Why Concurrency Control is needed:

- **The Lost Update Problem.**

This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

- **The Temporary Update (or Dirty Read) Problem.**

This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4). The updated item is accessed by another transaction before it is changed back to its original value.

Introduction to Transaction Processing (8)

Why Concurrency Control is needed (cont.):

- **The Incorrect Summary Problem .**

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

FIGURE 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem.

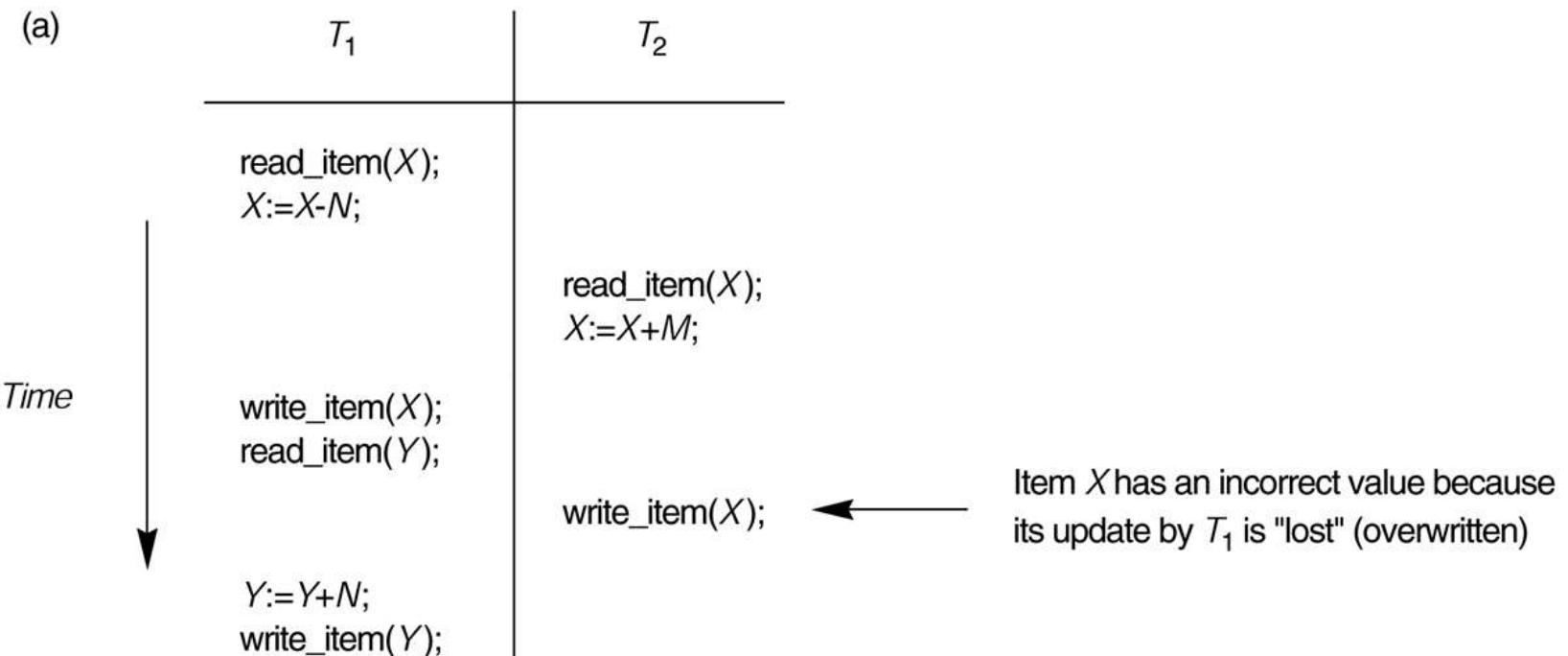
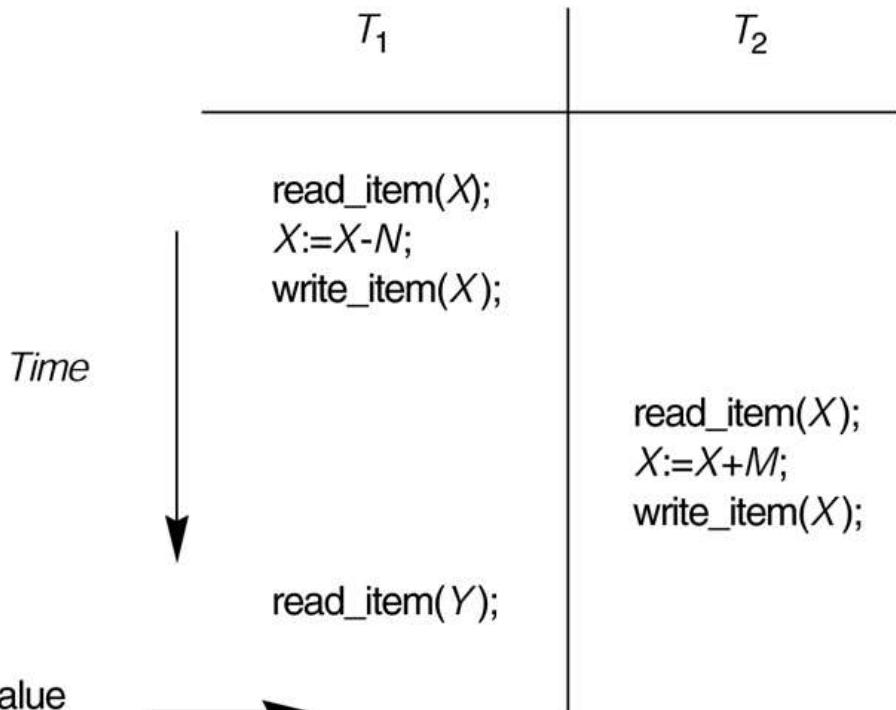


FIGURE 17.3 (continued)

Some problems that occur when concurrent execution is uncontrolled. (b) The temporary update problem.

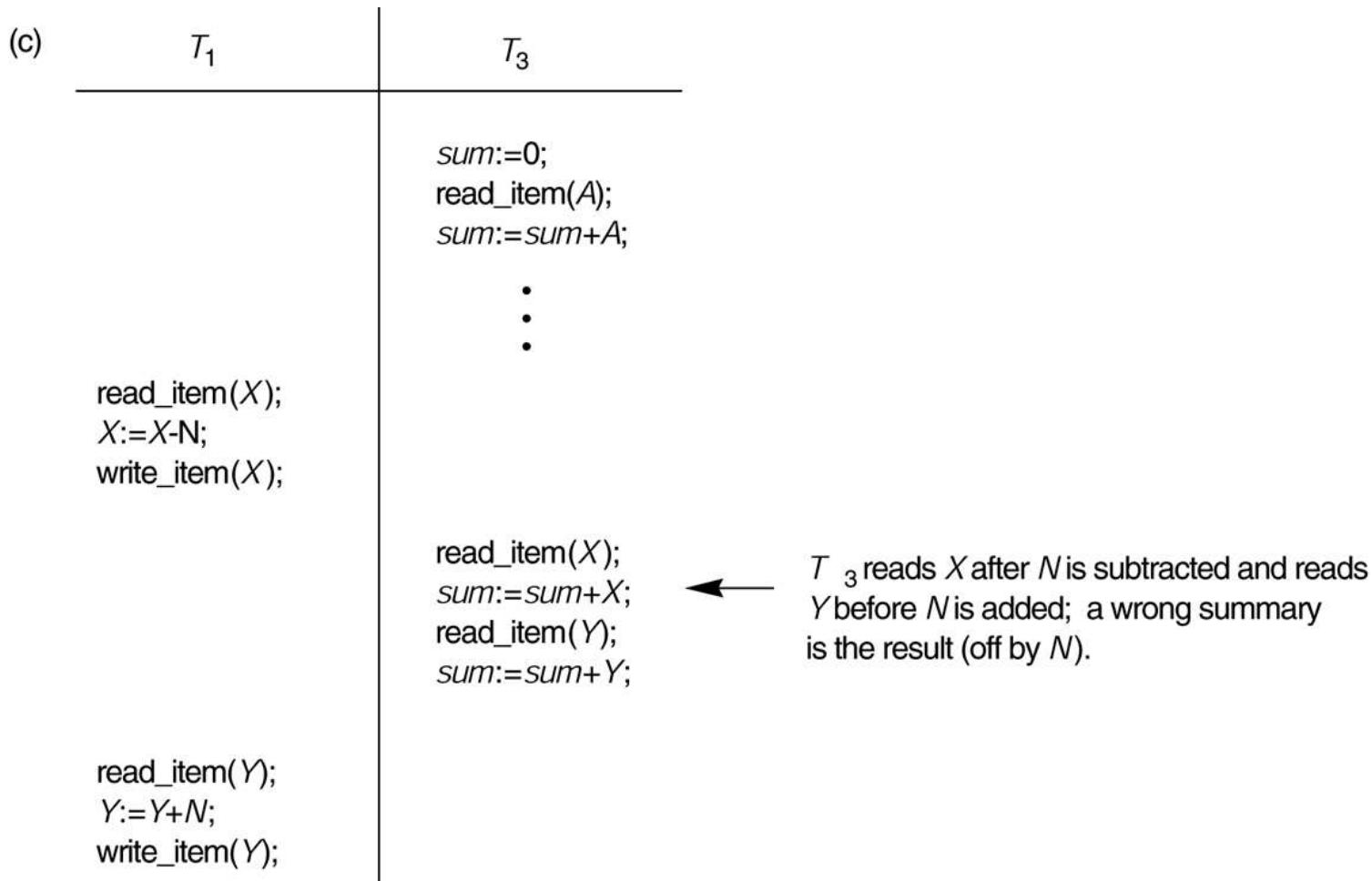
(b)



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

FIGURE 17.3 (continued)

Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



Introduction to Transaction Processing (11)

Why recovery is needed:

(What causes a Transaction to fail)

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Introduction to Transaction Processing (12)

Why recovery is needed (cont.):

3. Local errors or exception conditions detected by the transaction:

- certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.
- a programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement: The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (see Chapter 18).

Introduction to Transaction Processing (13)

Why recovery is needed (cont.):

5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

2 Transaction and System Concepts (1)

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

Transaction states:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

Transaction and System Concepts (2)

Recovery manager keeps track of the following operations:

- **begin_transaction**: This marks the beginning of transaction execution.
- **read or write**: These specify read or write operations on the database items that are executed as part of a transaction.
- **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution. At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

Transaction and System Concepts (3)

Recovery manager keeps track of the following operations (cont):

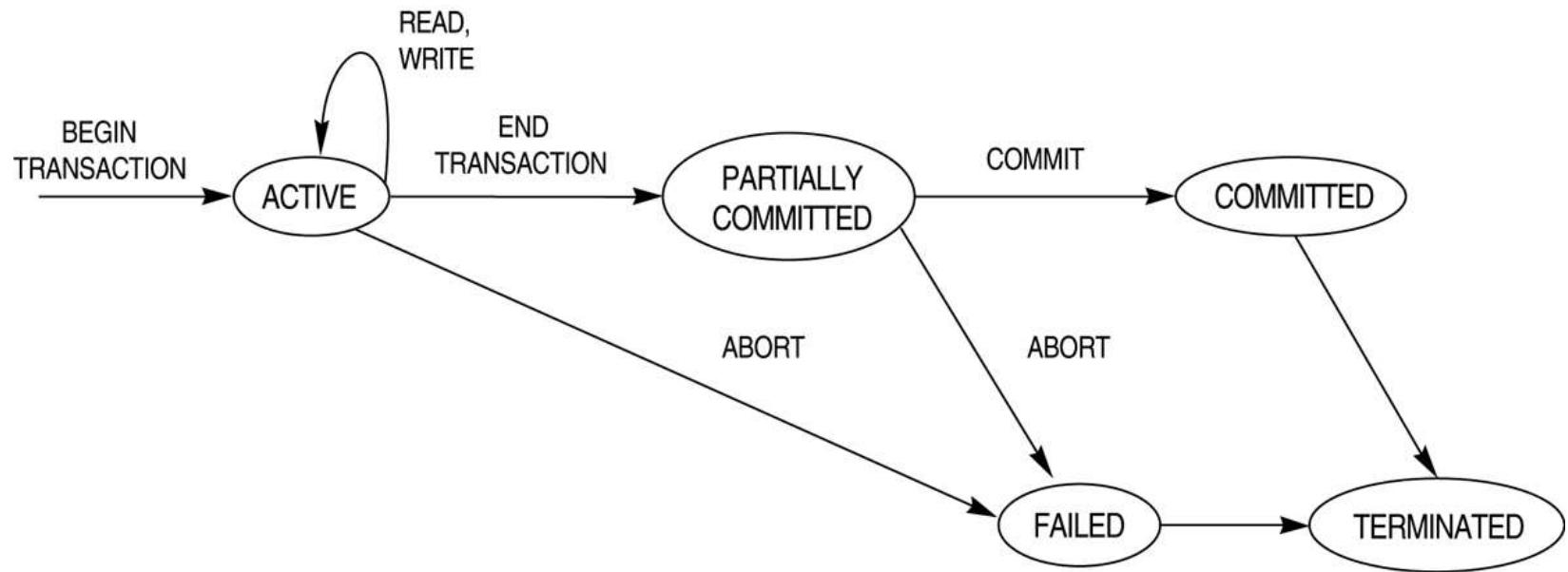
- **commit_transaction:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- **rollback (or abort):** This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone*.

Transaction and System Concepts (4)

Recovery techniques use the following operators:

- **undo:** Similar to rollback except that it applies to a single operation rather than to a whole transaction.
- **redo:** This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

FIGURE 17.4
State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts (6)

The System Log

- **Log or Journal** : The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.
- T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

Transaction and System Concepts (7)

The System Log (cont):

Types of log record:

1. [start_transaction,T]: Records that transaction T has started execution.
2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Records that transaction T has read the value of database item X.
4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Records that transaction T has been aborted.

Transaction and System Concepts (8)

The System Log (cont):

- protocols for recovery that avoid cascading rollbacks do not require that read operations be written to the system log, whereas other protocols require these entries for recovery.
- strict protocols require simpler write entries that do not include new_value (see Section 17.4).

Transaction and System Concepts (9)

Recovery using log records:

If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19.

1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their `old_values`.
2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their `new_values`.

Transaction and System Concepts (10)

Commit Point of a Transaction:

- **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:** Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Transaction and System Concepts (11)

Commit Point of a Transaction (cont):

- **Redoing transactions:** Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost.)
- **Force writing a log:** *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

3 Desirable Properties of Transactions (1)

ACID properties:

- **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another.

Desirable Properties of Transactions (2)

ACID properties (cont.):

- **Isolation:** A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary (see Chapter 21).
- **Durability or permanency:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

4 Characterizing Schedules based on Recoverability (1)

- **Transaction schedule or history:** When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).
- **A schedule (or history) S of n transactions T₁, T₂, ..., T_n :**
It is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operations of T_i in S must appear in the same order in which they occur in T_i. Note, however, that operations from other transactions T_j can be interleaved with the operations of T_i in S.

Characterizing Schedules based on Recoverability (2)

Schedules classified on recoverability:

- **Recoverable schedule:** One where no transaction needs to be rolled back.
A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
- **Cascadeless schedule:** One where every transaction reads only the items that are written by committed transactions.

Schedules requiring cascaded rollback: A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

Characterizing Schedules based on Recoverability (3)

Schedules classified on recoverability (cont.):

- **Strict Schedules:** A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

5 Characterizing Schedules based on Serializability (1)

- **Serial schedule:** A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise, the schedule is called **nonserial schedule**.
- **Serializable schedule:** A schedule S is **serializable** if it is equivalent to some serial schedule of the same n transactions.

Characterizing Schedules based on Serializability (2)

- **Result equivalent:** Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent:** Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable:** A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S' .

Characterizing Schedules based on Serializability (3)

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
 - It will leave the database in a consistent state.
 - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

Characterizing Schedules based on Serializability (4)

- Serializability is hard to check.
 - Interleaving of operations occurs in an operating system through some scheduler
 - Difficult to determine beforehand how the operations in a schedule will be interleaved.

Characterizing Schedules based on Serializability (5)

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends. Hence, we reduce the problem of checking the whole schedule to checking only a *committed project* of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
 - Use of locks with two phase locking

Characterizing Schedules based on Serializability (6)

- **View equivalence:** A less restrictive definition of equivalence of schedules
- **View serializability:** definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Characterizing Schedules based on Serializability (7)

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

1. The same set of transactions participates in S and S' , and S and S' include the same operations of those transactions.
2. For any operation $R_i(X)$ of T_i in S , if the value of X read by the operation has been written by an operation $W_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $R_i(X)$ of T_i in S' .
3. If the operation $W_k(Y)$ of T_k is the last operation to write item Y in S , then $W_k(Y)$ of T_k must also be the last operation to write item Y in S' .

Characterizing Schedules based on Serializability (8)

The premise behind view equivalence:

- 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 view**”: the read operations are said to see the *the same view* in both schedules.

Characterizing Schedules based on Serializability (9)

Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Characterizing Schedules based on Serializability (10)

Relationship between view and conflict equivalence (cont):

Consider the following schedule of three transactions

T1: r1(X), w1(X); T2: w2(X); and T3: w3(X);

Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.

Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule.

Characterizing Schedules based on Serializability (11)

Testing for conflict serializability

Algorithm 17.1:

1. Looks at only read_Item (X) and write_Item (X) operations
2. Constructs a precedence graph (serialization graph) - a graph with directed edges
3. An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
4. The schedule is serializable if and only if the precedence graph has no cycles.

FIGURE 17.7

Constructing the precedence graphs for schedules *A* and *D* from Figure 17.5 to test for conflict serializability. (a) Precedence graph for serial schedule *A*. (b) Precedence graph for serial schedule *B*. (c) Precedence graph for schedule *C* (not serializable). (d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

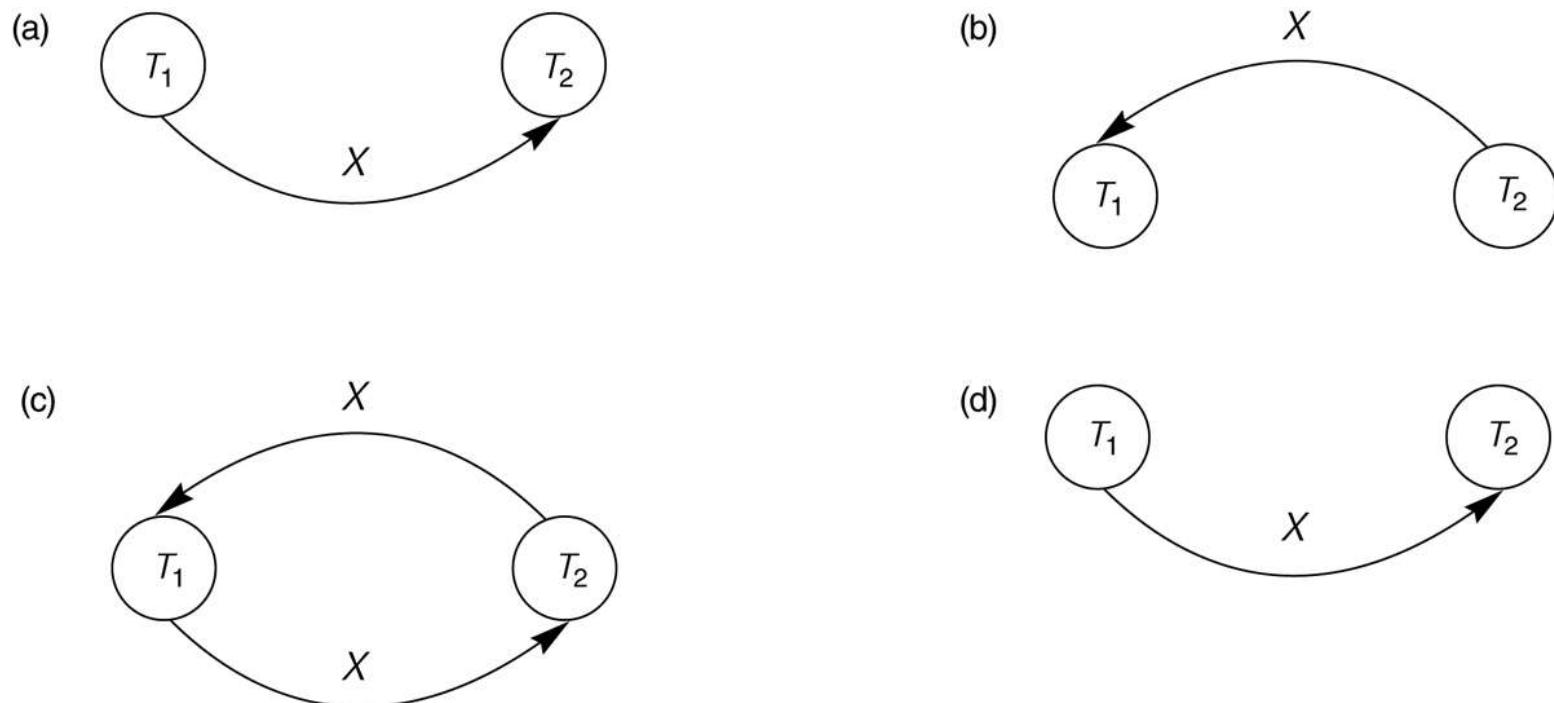


FIGURE 17.8

Another example of serializability testing. (a) The READ and WRITE operations of three transactions T_1 , T_2 , and T_3 .

(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);

FIGURE 17.8 (continued)

Another example of serializability testing. (b) Schedule E.

(b)	transaction T_1	transaction T_2	transaction T_3
<i>Time</i>		read_item (Z); read_item (Y); write_item (Y);	
	read_item (X); write_item (X);	read_item (X); write_item (X);	read_item (Y); read_item (Z); write_item (Y); write_item (Z);
	read_item (Y); write_item (Y);		

Schedule E

FIGURE 17.8 (continued)

Another example of serializability testing. (c) Schedule F.

(c)	transaction T_1	transaction T_2	transaction T_3
<i>Time</i>	↓		
	read_item (X); write_item (X);	read_item (Z);	read_item (Y); read_item (Z);
	read_item (Y); write_item (Y);	read_item (Y); write_item (Y); read_item (X); write_item (X);	write_item (Y); write_item (Z);

Schedule F

Characterizing Schedules based on Serializability (14)

Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly. Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

Characterizing Schedules based on Serializability (15)

Other Types of Equivalence of Schedules (cont.)

Example: bank credit / debit transactions on a given item are **separable** and **commutative**.

Consider the following schedule S for the two transactions:

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

Using conflict serializability, it is not **serializable**.

However, if it came from a (read,update, write) sequence as follows:

r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20;r1(Y);

Y := Y + 10; w1(Y); r2(X); X := X + 20; (X);

Sequence explanation: debit, debit, credit, credit.

It is a correct schedule for the given semantics

6 Transaction Support in SQL2 (1)

- A single SQL statement is always considered to be atomic. Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Transaction Support in SQL2 (2)

Characteristics specified by a SET TRANSACTION statement in SQL2:

- **Access mode:** READ ONLY or READ WRITE. The default is READ WRITE unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size n,** specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area. (Supply user feedback information)

Transaction Support in SQL2 (3)

Characteristics specified by a SET TRANSACTION statement in SQL2 (cont.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability. However, if any transaction executes at a lower level, then serializability may be violated.

Transaction Support in SQL2 (4)

Potential problem with lower isolation levels:

- **Dirty Read:** Reading a value that was written by a transaction which failed.
- **Nonrepeatable Read:** Allowing another transaction to write a new value between multiple reads of one transaction.

A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value. Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

Transaction Support in SQL2 (5)

Potential problem with lower isolation levels (cont.):

- **Phantoms:** New rows being read using the same read with a condition.
A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause. Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1. If T1 is repeated, then T1 will see a row that previously did not exist, called a **phantom**.

Transaction Support in SQL2 (6)

Sample SQL transaction:

```
EXEC SQL whenever sqlerror go to UNDO;
```

```
EXEC SQL SET TRANSACTION
```

```
    READ WRITE
```

```
    DIAGNOSTICS 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: ...
```

Transaction Support in SQL2 (7)

Possible violation of serializability:

Type of Violation

Isolation level	Dirty read	nonrepeatable read	phantom read
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no

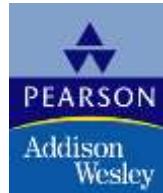
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 18

Concurrency Control Techniques



Chapter 18 Outline

Databases Concurrency Control

- 1 Purpose of Concurrency Control
- 2 Two-Phase locking
- 5 Limitations of CCMs
- 6 Index Locking
- 7 Lock Compatibility Matrix
- 8 Lock Granularity

Database Concurrency Control

1 Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example: In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Database Concurrency Control

Two-Phase Locking Techniques

Locking is an operation which secures (a) permission to Read or (b) permission to Write a data item for a transaction. Example: Lock (X). Data item X is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item. Example: Unlock (X). Data item X is made available to all other transactions.

Lock and Unlock are Atomic operations.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Two locks modes (a) shared (read) and (b) exclusive (write).

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Conflict matrix

		Read	Write
Read	Read	Y	N
	Write	N	N

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock Manager: Managing locks on data items.

Lock table: Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Database requires that all transactions should be well-formed. A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)
    then LOCK (X)  $\leftarrow$  1 (*lock the item*)
    else begin
        wait (until lock (X) = 0) and
        the lock manager wakes up the transaction);
    goto B
end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

LOCK (X) \leftarrow 0 (*unlock the item*)

if any transactions are waiting then

 wake up one of the waiting the transactions;

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the read operation:

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 $\text{LOCK}(X) \leftarrow \text{"read-locked"}$ 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;

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the write lock operation:

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 $\text{LOCK}(X) \leftarrow \text{"read-locked"}$ 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;

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

```
if LOCK (X) = “write-locked” then
    begin LOCK (X) ← “unlocked”;
        wakes up one of the transactions, if any
    end
else if LOCK (X) ← “read-locked” then
    begin
        no_of_reads (X) ← no_of_reads (X) -1
        if no_of_reads (X) = 0 then
            begin
                LOCK (X) = “unlocked”;
                wake up one of the transactions, if any
            end
    end;
end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock conversion

Lock upgrade: existing read lock to write lock

if T_i has a read-lock (X) and T_j has no read-lock (X) ($i \neq j$) then
 convert read-lock (X) to write-lock (X)
else
 force T_i to wait until T_j unlocks X

Lock downgrade: existing write lock to read lock

T_i has a write-lock (X) (*no transaction can have any lock on X*)
 convert write-lock (X) to read-lock (X)

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two Phases: (a) Locking (Growing) (b) Unlocking (Shrinking).

Locking (Growing) Phase: A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase: A transaction unlocks its locked data items one at a time.

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

<u>T1</u>	<u>T2</u>	<u>Result</u>
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);	Initial values: X=20; Y=30 Result of serial execution T1 followed by T2 X=50, Y=80. Result of serial execution T2 followed by T1 X=70, Y=50

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2	<u>Result</u>
read_lock (Y); read_item (Y); unlock (Y);	read_lock (X); read_item (X); unlock (X); write_lock (Y); read_item (Y); $Y:=X+Y;$ write_item (Y); unlock (Y);	X=50; Y=50 Nonserializable because it. violated two-phase policy.
write_lock (X); read_item (X); $X:=X+Y;$ write_item (X); unlock (X);		

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1

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

T'2

```
read_lock (X);  
read_item (X);  
Write_lock (Y);  
unlock (X);  
read_item (Y);  
Y:=X+Y;  
write_item (Y);  
unlock (Y);
```

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two-phase policy generates two locking algorithms (a) Basic and (b) Conservative.

Conservative: Prevents deadlock by locking all desired data items before transaction begins execution.

Basic: Transaction locks data items incrementally. This may cause deadlock which is dealt with.

Strict: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock

T'1

```
read_lock (Y);  
read_item (Y);
```

```
write_lock (X);  
(waits for X)
```

Deadlock (T'1 and T'2)

T'2

```
read_lock (X);  
read_item (Y);
```

```
write_lock (Y);  
(waits for Y)
```

T1 and T2 did follow two-phase policy but they are deadlock

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock prevention

A transaction locks all data items it refers to before it begins execution. This way of locking prevents deadlock since a transaction never waits for a data item. The conservative two-phase locking uses this approach.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock detection and resolution

In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.

A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: T_i waits for T_j waits for T_k waits for T_i or T_j occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock avoidance

There are many variations of two-phase locking algorithm. Some avoid deadlock by not letting the cycle to complete. That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction. Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Database Concurrency Control

Dealing with Deadlock and Starvation

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.

Database Concurrency Control

Timestamp based concurrency control algorithm

Timestamp

A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.

Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Database Concurrency Control

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
 - a. If $\text{read_TS}(X) > \text{TS}(T)$ or if $\text{write_TS}(X) > \text{TS}(T)$, then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
 - b. If the condition in part (a) does not exist, then execute write_item(X) of T and set $\text{write_TS}(X)$ to $\text{TS}(T)$.
2. Transaction T issues a read_item(X) operation:
 - a. If $\text{write_TS}(X) > \text{TS}(T)$, then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
 - b. If $\text{write_TS}(X) \leq \text{TS}(T)$, then execute read_item(X) of T and set $\text{read_TS}(X)$ to the larger of $\text{TS}(T)$ and the current $\text{read_TS}(X)$.

Database Concurrency Control

Timestamp based concurrency control algorithm

Strict Timestamp Ordering

1. Transaction T issues a `write_item(X)` operation:
 - a. If $TS(T) > \text{read_TS}(X)$, 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:
 - a. If $TS(T) > \text{write_TS}(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Database Concurrency Control

Timestamp based concurrency control algorithm

Thomas's Write Rule

1. If $\text{read_TS}(X) > \text{TS}(T)$ then abort and roll-back T and reject the operation.
2. If $\text{write_TS}(X) > \text{TS}(T)$, then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. 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 $\text{TS}(T)$.

Database Concurrency Control

Multiversion concurrency control techniques

Concept

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 effect: 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.

Database Concurrency Control

Multiversion technique based on timestamp ordering

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.

Database Concurrency Control

Multiversion technique based on timestamp ordering

Assume X_1, X_2, \dots, X_n are the versions of a data item X created by a write operation 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.

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read_TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).

If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read_TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

1. If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read_TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).
2. If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read_TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

Rule 2 guarantees that a read will never be rejected.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks Concept

Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T .

This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X .

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Steps

1. X is the committed version of a data item.
2. T creates a second version X' after obtaining a write lock on X.
3. Other transactions continue to read X.
4. T is ready to commit so it obtains a certify lock on X'.
5. The committed version X becomes X'.
6. T releases its certify lock on X', which is X now.

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

read/write locking scheme

	Read	Write	Certify
Read	yes	no	no
Write	no	no	no
Certify	no	no	no

read/write/certify locking scheme

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Note

In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently. This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes. It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.

Three phases:

Read phase: A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

Validation phase: Serializability is checked before transactions write their updates to the database.

This phase for T_i checks that, for each transaction T_j that is either committed or is in its validation phase, one of the following conditions holds:

1. T_j completes its write phase before T_i starts its read phase.
2. T_i starts its write phase after T_j completes its write phase, and the `read_set` of T_i has no items in common with the `write_set` of T_j
3. Both the `read_set` and `write_set` of T_i have no items in common with the `write_set` of T_j , and T_j completes its read phase.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

When validating T_i , the first condition is checked first for each transaction T_j , since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and T_i is aborted.

Write phase: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

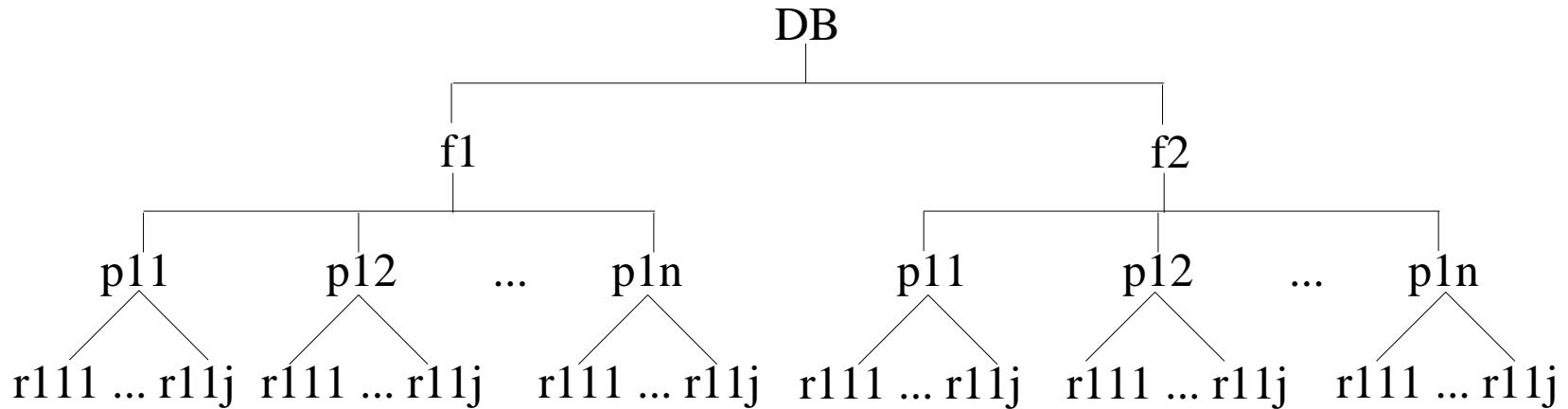
A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation). Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for 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.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

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 nodes(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).

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

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

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

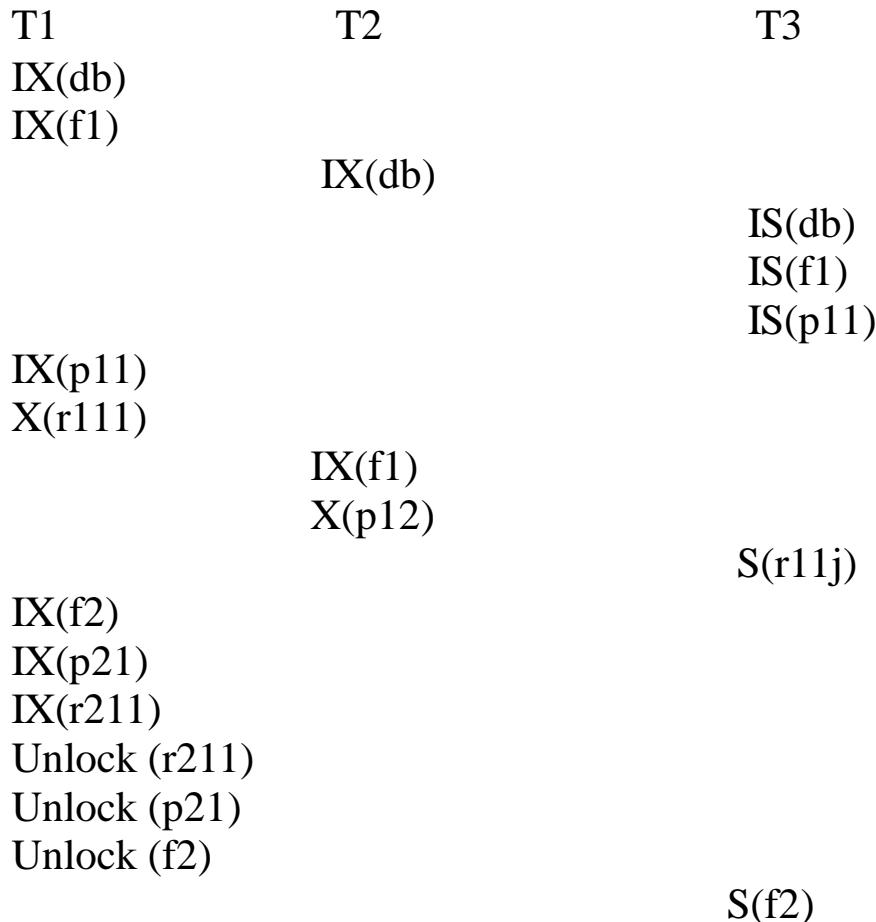
The set of rules which must be followed for producing serializable schedule are

1. The lock compatibility must adhered to.
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 IX 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.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution:



Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution (continued):

T1

unlock(r111)
unlock(p11)
unlock(f1)
unlock(db)

T2

unlock(p12)
unlock(f1)
unlock(db)

T3

unlock (r111j)
unlock (p11)
unlock (f1)
unlock(f2)
unlock(db)

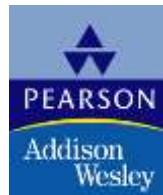
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 19

Database Recovery Techniques



Chapter 19 Outline

Databases Recovery

- 1 Purpose of Database Recovery
- 2 Types of Failure
- 3 Transaction Log
- 4 Data Updates
- 5 Data Caching
- 6 Transaction Roll-back (Undo) and Roll-Forward
- 7 Checkpointing
- 8 Recovery schemes
- 9 ARIES Recovery Scheme
- 10 Recovery in Multidatabase System

Database Recovery

1 Purpose of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).

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.

Database Recovery

2 Types of Failure

The database may become unavailable for use due to

- Transaction failure: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
- System failure: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- Media failure: Disk head crash, power disruption, etc.

Database Recovery

3 Transaction Log

For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required. These values and other information is stored in a sequential file called Transaction log. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Database Recovery

4 Data Update

- **Immediate Update:** As soon as a data item is modified in cache, the disk copy is updated.
- **Deferred Update:** All modified data items in the cache are written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- **Shadow update:** The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- **In-place update:** The disk version of the data item is overwritten by the cache version.

Database Recovery

5 Data Caching

Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk. The flushing is controlled by **Modified** and **Pin-Unpin** bits.

Pin-Unpin: Instructs the operating system not to flush the data item.

Modified: Indicates the AFIM of the data item.

Database Recovery

6 Transaction Roll-back (Undo) and Roll-Forward (Redo)

To maintain atomicity, a transaction's operations are **redone** or **undone**.

Undo: Restore all BFIMs on to disk (Remove all AFIMs).

Redo: Restore all AFIMs on to disk.

Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.

Database Recovery

Roll-back

We show the process of roll-back with the help of the following three transactions T1, and T2 and T3.

T1

read_item (A)
read_item (D)
write_item (D)

T2

read_item (B)
write_item (B)
read_item (D)
write_item (A)

T3

read_item (C)
write_item (B)
read_item (A)
write_item (A)

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.

A	B	C	D
30	15	40	20

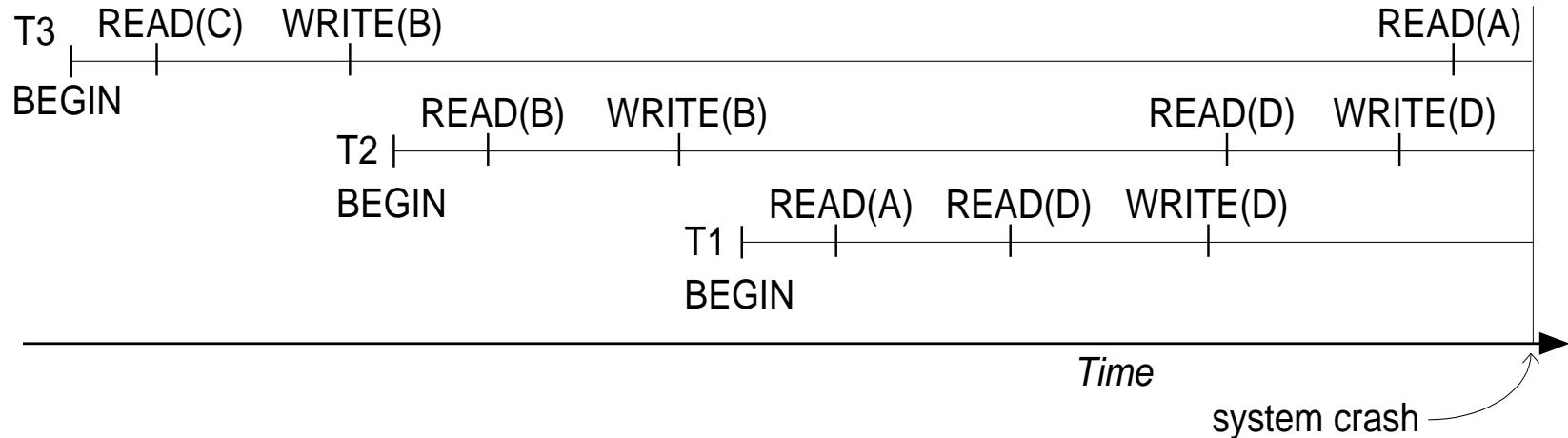
- [start_transaction, T3]
- [read_item, T3, C]
- * [write_item, T3, B, 15, 12] 12
- [start_transaction, T2]
- [read_item, T2, B]
- ** [write_item, T2, B, 12, 18] 18
- [start_transaction, T1]
- [read_item, T1, A]
- [read_item, T1, D]
- [write_item, T1, D, 20, 25] 25
- [read_item, T2, D]
- ** [write_item, T2, D, 25, 26] 26
- [read_item, T3, A]

---- system crash ----

- * T3 is rolled back because it did not reach its commit point.
- ** T2 is rolled back because it reads the value of item B written by T3.

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.



Illustrating cascading roll-back

Database Recovery

Write-Ahead Logging

When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging (WAL)** protocol. WAL states that

For Undo: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).

For Redo: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

Database Recovery

7 Checkpointing

Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps defines a checkpoint operation:

1. Suspend execution of transactions temporarily.
2. Force write modified buffer data to disk.
3. Write a [checkpoint] record to the log, save the log to disk.
4. Resume normal transaction execution.

During recovery **redo** or **undo** is required to transactions appearing after [checkpoint] record.

Database Recovery

Steal/No-Steal and Force/No-Force

Possible ways for flushing database cache to database disk:

Steal: Cache can be flushed before transaction commits.

No-Steal: Cache cannot be flushed before transaction commit.

Force: Cache is immediately flushed (forced) to disk.

No-Force: Cache is deferred until transaction commits.

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) and No-Steal/Force (No-undo/No-redo).

Database Recovery

8 Recovery Scheme

Deferred Update (No Undo/Redo)

The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

Database Recovery

Deferred Update in a single-user system

There is no concurrent data sharing in a single user system. The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

Database Recovery

Deferred Update in a single-user system

(a)	T1	T2
	read_item (A)	read_item (B)
	read_item (D)	write_item (B)
	write_item (D)	read_item (D)
		write_item (A)

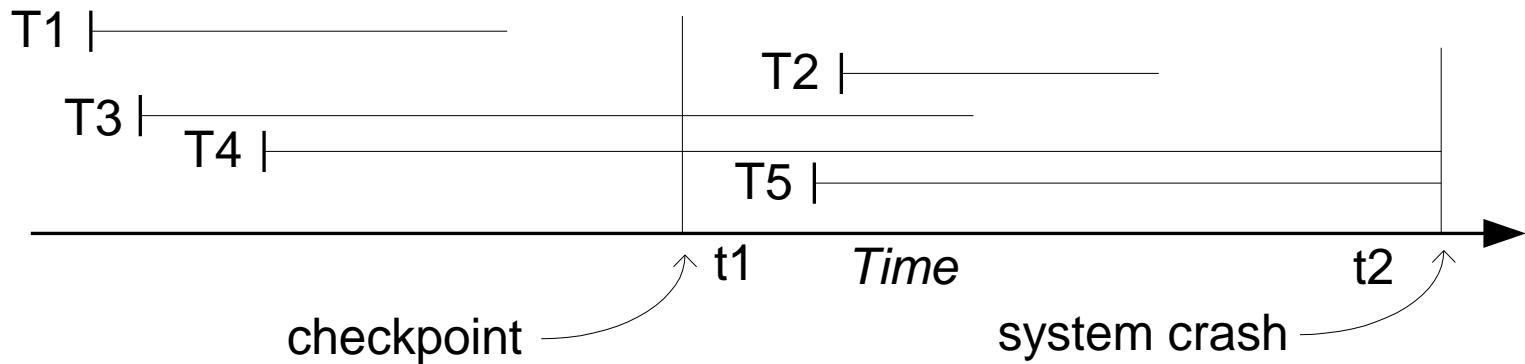
(b)	[start_transaction, T1]	
	[write_item, T1, D, 20]	
	[commit T1]	
	[start_transaction, T1]	
	[write_item, T2, B, 10]	
	[write_item, T2, D, 25]	← system crash

The [write_item, ...] operations of T1 are redone.
T2 log entries are ignored by the recovery manager.

Database Recovery

Deferred Update with concurrent users

This environment requires some concurrency control mechanism to guarantee isolation property of transactions. In a system recovery transactions which were recorded in the log after the last checkpoint were redone. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.



Recovery in a concurrent users environment.

Database Recovery

Deferred Update with concurrent users

- (a) T1 T2 T3 T4
read_item (A) read_item (B) read_item (A) read_item (B)
read_item (D) write_item (B) write_item (A) write_item (B)
write_item (D) read_item (D) read_item (C) read_item (A)
 write_item (D) write_item (C) write_item (A)
- (b) [start_transaction, T1]
[write_item, T1, D, 20]
[commit, T1]
[checkpoint]
[start_transaction, T4]
[write_item, T4, B, 15]
[write_item, T4, A, 20]
[commit, T4]
[start_transaction T2]
[write_item, T2, B, 12]
[start_transaction, T3]
[write_item, T3, A, 30]
[write_item, T2, D, 25] ← system crash

T2 and T3 are ignored because they did not reach their commit points.
T4 is redone because its commit point is after the last checkpoint.

Database Recovery

Deferred Update with concurrent users

Two tables are required for implementing this protocol:

Active table: All active transactions are entered in this table.

Commit table: Transactions to be committed are entered in this table.

During recovery, all transactions of the commit table are redone and all transactions of active tables are ignored since none of their AFIMs reached the database. It is possible that a commit table transaction may be redone twice but this does not create any inconsistency because of a redone is “idempotent”, that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

Database Recovery

Recovery Techniques Based on Immediate Update

Undo/No-redo Algorithm

In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits. For this reason the recovery manager undoes all transactions during recovery. No transaction is redone. It is possible that a transaction might have completed execution and ready to commit but this transaction is also undone.

Database Recovery

Recovery Techniques Based on Immediate Update

Undo/Redo Algorithm (Single-user environment)

Recovery schemes of this category apply undo and also redo for recovery. In a single-user environment no concurrency control is required but a log is maintained under WAL. Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table. The recovery manager performs:

1. Undo of a transaction if it is in the active table.
2. Redo of a transaction if it is in the commit table.

Database Recovery

Recovery Techniques Based on Immediate Update

Undo/Redo Algorithm (Concurrent execution)

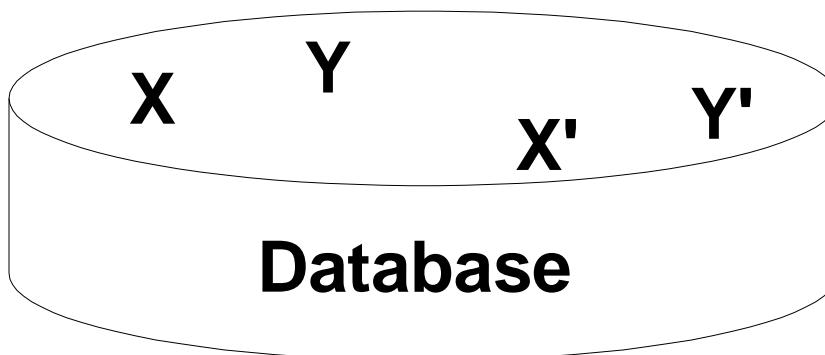
Recovery schemes of this category applies undo and also redo to recover the database from failure. In concurrent execution environment a concurrency control is required and log is maintained under WAL. Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager checkpointing is used. The recovery performs:

1. Undo of a transaction if it is in the active table.
2. Redo of a transaction if it is in the commit table.

Database Recovery

Shadow Paging

The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.

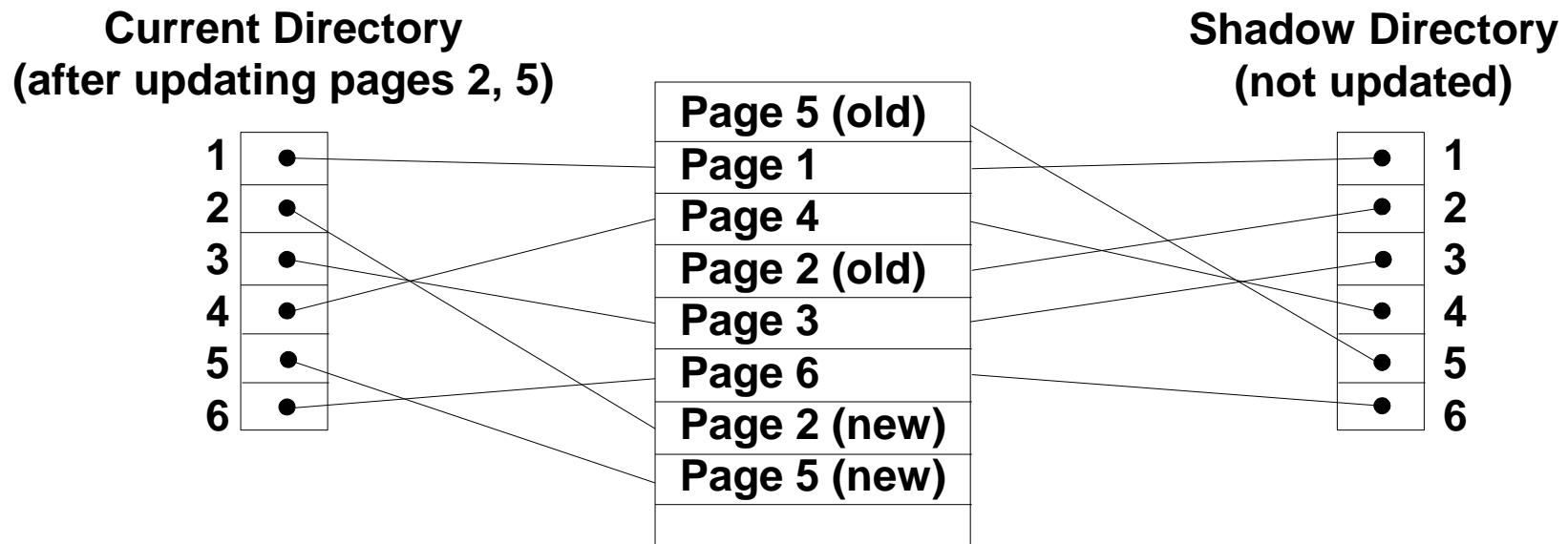


X and Y: Shadow copies of data items
X` and Y`: Current copies of data items

Database Recovery

Shadow Paging

To manage access of data items by concurrent transactions two directories (current and shadow) are used. The directory arrangement is illustrated below. Here a page is a data item.



Database Recovery

9 The ARIES Recovery Algorithm

The ARIES Recovery Algorithm is based on:

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 database state when the crash occurred.
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.

Database Recovery

The ARIES Recovery Algorithm

The ARIES recovery algorithm consists of three steps:

1. Analysis: step identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of crash. The appropriate point in the log where redo is to start is also determined.
2. Redo: 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.

Database Recovery

The ARIES Recovery Algorithm

The Log and Log Sequence Number (LSN)

A log record is written for (a) data update, (b) transaction commit, (c) transaction abort, (d) undo, and (e) transaction end. In the case of undo a compensating log record is written.

A unique LSN is associated with every log record. LSN increases monotonically and indicates the disk address of the log record it is associated with. In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.

A log record stores (a) the previous LSN of that transaction, (b) the transaction ID, and (c) the type of log record.

Database Recovery

The ARIES Recovery Algorithm

The Log and Log Sequence Number (LSN)

A log record stores:

1. Previous LSN of that transaction: It links the log record of each transaction. It is like a back pointer points to the previous record of the same transaction.
2. Transaction ID
3. Type of log record.

For a write operation the following additional information is logged:

4. Page ID for the page that includes the item
5. Length of the updated item
6. Its offset from the beginning of the page
7. BFIM of the item
8. AFIM of the item

Database Recovery

The ARIES Recovery Algorithm

The Transaction table and the Dirty Page table

For efficient recovery following tables are also stored in the log during checkpointing:

Transaction table: Contains an entry for each active transaction, with information such as transaction ID, transaction status and the LSN of the most recent log record for the transaction.

Dirty Page table: Contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

Database Recovery

The ARIES Recovery Algorithm

Checkpointing

A checkpointing does the following:

1. Writes a *begin_checkpoint* record in the log
2. Writes an *end_checkpoint* record in the log. With this record the contents of transaction table and dirty page table are appended to the end of the log.
3. Writes the LSN of the *begin_checkpoint* record to a special file. This special file is accessed during recovery to locate the last checkpoint information.

To reduce the cost of checkpointing and allow the system to continue to execute transactions, ARIES uses “fuzzy checkpointing”.

Database Recovery

The ARIES Recovery Algorithm

The following steps are performed for recovery

1. Analysis phase: Start at the begin_checkpoint record and proceed to the end_checkpoint record. Access transaction table and dirty page table are appended to the end of the log. Note that during this phase some other log records may be written to the log and transaction table may be modified. The analysis phase compiles the set of redo and undo to be performed and ends.
2. Redo phase: Starts from the point in the log up to where all dirty pages have been flushed, and move forward to the end of the log. Any change that appears in the dirty page table is redone.
3. Undo phase: Starts from the end of the log and proceeds backward while performing appropriate undo. For each undo it writes a compensating record in the log.

The recovery completes at the end of undo phase.

Database Recovery

An example of the working of ARIES scheme

(a)

<u>LSN</u>	<u>LAST-LSN</u>	<u>TRAN-ID</u>	<u>TYPE</u>	<u>PAGE-ID</u>	<u>Other Info.</u>
1	0	T1	update	C	----
2	0	T2	update	B	----
3	1	T1	commit		----
4	begin checkpoint				
5	end checkpoint				
6	0	T3	update	A	----
7	2	T2	update	C	----
8	7	T2	commit		----

(b)

TRANSACTION TABLE			DIRTY PAGE TABLE	
<u>TRANSACTION ID</u>	<u>LAST LSN</u>	<u>STATUS</u>	<u>PAGE ID</u>	<u>LSN</u>
T1	3	commit	C	1
T2	2	in progress	B	2

(c)

TRANSACTION TABLE			DIRTY PAGE TABLE	
<u>TRANSACTION ID</u>	<u>LAST LSN</u>	<u>STATUS</u>	<u>PAGE ID</u>	<u>LSN</u>
T1	3	commit	C	1
T2	8	commit	B	2
T3	6	in progress	A	6

Database Recovery

10 Recovery in multidatabase system

A multidatabase system is a special distributed database system where one node may be running relational database system under Unix, another may be running object-oriented system under window and so on. A transaction may run in a distributed fashion at multiple nodes. In this execution scenario the transaction commits only when all these multiple nodes agree to commit individually the part of the transaction they were executing. This commit scheme is referred to as “*two-phase commit*” (2PC). If any one of these nodes fails or cannot commit the part of the transaction, then the transaction is aborted. Each node recovers the transaction under its own recovery protocol.

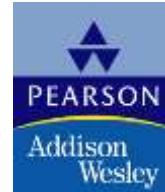
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 21

Object Database Standards, Languages, and Design



Chapter 21 Outline

- 21.1 Overview of the Object Model ODMG**
- 21.2 The Object Definition Language DDL**
- 21.3 The Object Query Language OQL**
- 21.4 Overview of C++ Binding**
- 21.5 Object Database Conceptual Model**
- 21.6 Summary**

Chapter Objectives

- Discuss the importance of standards (e.g., portability, interoperability)
- Introduce Object Data Management Group (ODMG): object model, object definition language (ODL), object query language (OQL)
- Present ODMG object binding to programming languages (e.g., C++)
- Present Object Database Conceptual Design

21.1 The Object Model of ODMG

- Provides a standard model for object databases
- Supports object definition via ODL
- Supports object querying via OQL
- Supports a variety of data types and type constructors

ODMG Objects and Literals

- The basic building blocks of the object model are
 - Objects
 - Literals
- An object has four characteristics
 1. Identifier: unique system-wide identifier
 2. Name: unique within a particular database and/or program; it is optional
 3. Lifetime: persistent vs transient
 4. Structure: specifies how object is constructed by the type constructor and whether it is an *atomic* object

ODMG Literals

- A literal has a current value but not an identifier
- Three types of literals
 1. *atomic*: predefined; basic data type values (e.g., short, float, boolean, char)
 2. *structured*: values that are constructed by type constructors (e.g., date, struct variables)
 3. *collection*: a collection (e.g., array) of values or objects

ODMG Interface Definition: An Example

- Note: interface is ODMG's keyword for class/type

```
interface Date:Object {  
    enum weekday{sun,mon,tue,wed,thu,fri,sat};  
    enum Month{jan,feb,mar,...,dec};  
    unsigned short year();  
    unsigned short month();  
    unsigned short day();  
    ...  
    boolean is_equal(in Date other_date);  
};
```

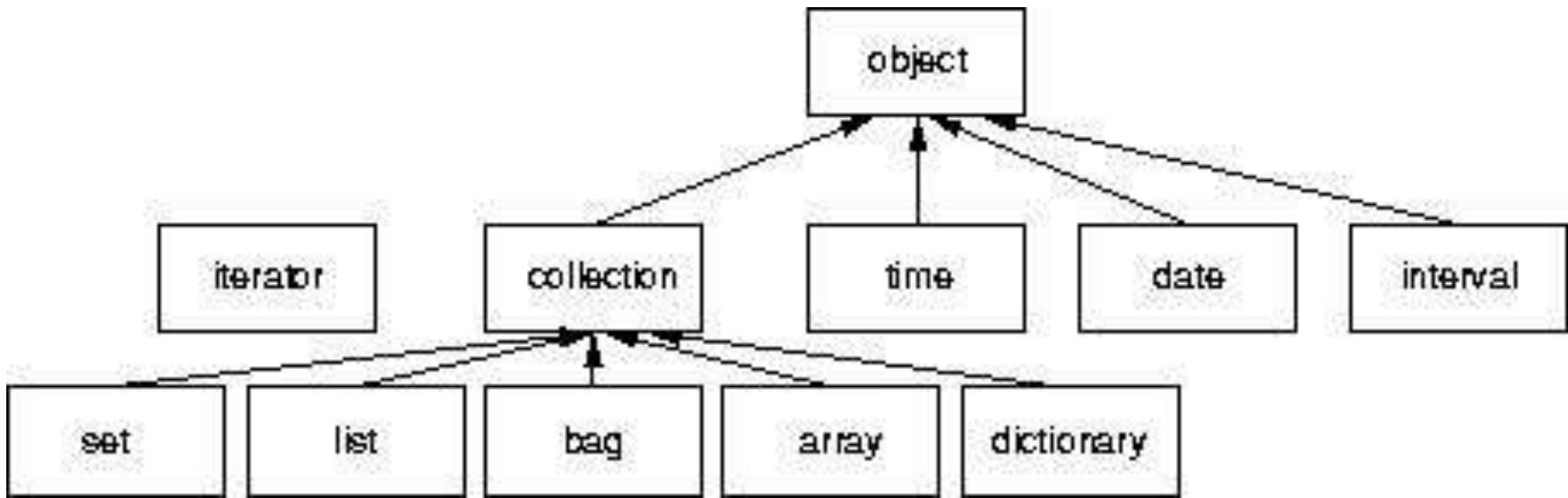
Built-in Interfaces for Collection Objects

- A collection object inherits the basic collection interface, for example:
 - cardinality()
 - is_empty()
 - insert_element()
 - remove_element()
 - contains_element()
 - create_iterator()

Collection Types

- Collection objects are further specialized into types like a set, list, bag, array, and dictionary
- Each collection type may provide additional interfaces, for example, a set provides:
 - `create_union()`
 - `create_difference()`
 - `is_subst_of()`
 - `is_superset_of()`
 - `is_proper_subset_of()`

Object Inheritance Hierarchy



Atomic Objects

- Atomic objects are user-defined objects and are defined via keyword `class`
- An example :

```
class Employee (extent all_employees key ssn) {  
    attribute string name;  
    attribute string ssn;  
    attribute short age;  
    relationship Dept works_for;  
    void reassign(in string new_name);  
}
```

Class Extents

- An ODMG object can have an extent defined via a class declaration
- Each extent is given a name and will contain all persistent objects of that class
- For Employee class, for example, the extent is called `all_employees`
- This is similar to creating an object of type `Set<Employee>` and making it persistent

Class Key

- A class key consists of one or more unique attributes
- For the Employee class, the key is ssn
Thus each employee is expected to have a unique ssn
- Keys can be composite, e.g.,
(key dnumber, dname)

Object Factory

- An object factory is used to generate individual objects via its operations
- An example:

```
interface ObjectFactory {  
    Object new ();  
};
```

- `new()` returns new objects with an `object_id`
- One can create their own factory interface by inheriting the above interface

Interface and Class Definition

- ODMG supports two concepts for specifying object types:
 - Interface
 - Class
- There are similarities and differences between interfaces and classes
- Both have behaviors (operations) and state (attributes and relationships)

ODMG Interface

- An interface is a specification of the abstract behavior of an object type
- State properties of an interface (i.e., its attributes and relationships) cannot be inherited from
- Objects cannot be instantiated from an interface

ODMG Class

- A class is a specification of abstract behavior and state of an object type
- A class is Instantiable
- Supports “extends” inheritance to allow both state and behavior inheritance among classes
- Multiple inheritance via “extends” is not allowed

21.2 Object Definition Language

- ODL supports semantics constructs of ODMG
- ODL is independent of any programming language
- ODL is used to create object specification (classes and interfaces)
- ODL is not used for database manipulation

ODL Examples (1)

A Very Simple Class

- A very simple, straightforward class definition (*all examples are based on the university schema presented in Chapter 4 and graphically shown on page 680*):

```
class Degree {  
    attribute string college;  
    attribute string degree;  
    attribute string year;  
};
```

ODL Examples (2)

A Class With Key and Extent

- A class definition with “extent”, “key”, and more elaborate attributes; still relatively straightforward

```
class Person (extent persons key ssn) {  
    attribute struct Pname {string fname ...} name;  
    attribute string ssn;  
    attribute date birthdate;  
  
    ...  
    short age();  
}
```

ODL Examples (3)

A Class With Relationships

- Note extends (inheritance) relationship
- Also note “inverse” relationship

```
Class Faculty extends Person (extent faculty) {  
    attribute string rank;  
    attribute float salary;  
    attribute string phone;  
  
    ...  
    relationship Dept works_in inverse Dept::has_faculty;  
    relationship set<GradStu> advises inverse GradStu::advisor;  
    void give_raise (in float raise);  
    void promote (in string new_rank);  
};
```

Inheritance via ":" – An Example

```
interface Shape {  
    attribute struct point {...} reference_point;  
    float perimeter ();  
    ...  
};  
  
class Triangle: Shape (extent triangles) {  
    attribute short side_1;  
    attribute short side_2;  
    ...  
};
```

21.3 Object Query Language

- OQL is DMG's query language
- OQL works closely with programming languages such as C++
- Embedded OQL statements return objects that are compatible with the type system of the host language
- OQL's syntax is similar to SQL with additional features for objects

Simple OQL Queries

- Basic syntax: select...from...where...

```
SELECT d.name  
FROM   d in departments  
WHERE  d.college = 'Engineering';
```

- An *entry point* to the database is needed for each query
- An extent name (e.g., departments in the above example) may serve as an entry point

Iterator Variables

- Iterator variables are defined whenever a collection is referenced in an OQL query
- Iterator `d` in the previous example serves as an iterator and ranges over each object in the collection
- Syntactical options for specifying an iterator:
 - `d in departments`
 - `departments d`
 - `departments as d`

Data Type of Query Results

- The data type of a query result can be any type defined in the ODMG model
- A query does not have to follow the select...from...where... format
- A persistent name on its own can serve as a query whose result is a reference to the persistent object, e.g., departments; whose type is set<Departments>

Path Expressions

- A *path expression* is used to specify a path to attributes and objects in an entry point
- A path expression starts at a persistent object name (or its iterator variable)
- The name will be followed by zero or more dot connected relationship or attribute names, e.g., departments.chair;

Views as Named Objects

- The *define* keyword in OQL is used to specify an identifier for a *named query*
- The name should be unique; if not, the results will replace an existing named query
- Once a query definition is created, it will persist until deleted or redefined
- A view definition can include parameters

An Example of OQL View

- A view to include students in a department who have a minor:

```
define has_minor(dept_name) as
select s
from s in students
where s.minor_in.dname=dept_name
```

- `has_minor` can now be used in queries

Single Elements from Collections

- An OQL query returns a collection
- OQL's element operator can be used to return a single element from a singleton collection that contains one element:

```
element (select d from d in departments)
where d.dname = 'Software Engineering');
```

- If d is empty or has more than one elements, an *exception* is raised

Collection Operators

- OQL supports a number of aggregate operators that can be applied to query results
- The aggregate operators include `min`, `max`, `count`, `sum`, and `avg` and operate over a collection
- `count` returns an integer; others return the same type as the collection type

An Example of an OQL Aggregate Operator

- To compute the average GPA of all seniors majoring in Business:

```
avg (select s.gpa from s in students  
      where s.class = 'senior' and  
            s.majors_in.dname = 'Business' );
```

Membership and Quantification

- OQL provides membership and quantification operators:
 - $(e \text{ in } c)$ is true if e is in the collection c
 - $(\text{for all } e \text{ in } c : b)$ is true if all e elements of collection c satisfy b
 - $(\text{exists } e \text{ in } c : b)$ is true if at least one e in collection c satisfies b

An Example of Membership

- To retrieve the names of all students who completed CS101:

```
select s.name.fname s.name.lname  
from s in students  
where 'CS101' in  
(select c.name from c in  
s.completed_sections.section.of_course)  
;
```

Ordered Collections

- Collections that are lists or arrays allow retrieving their *first*, *last*, and *ith* elements
- OQL provides additional operators for extracting a sub-collection and concatenating two lists
- OQL also provides operators for ordering the results

An Example of Ordered Operation

- To retrieve the last name of the faculty member who earns the highest salary:

```
first (select struct  
       (faculty: f.name.lastname, salary  
        f.salary)  
     from f in faculty  
    ordered by f.salary desc);
```

Grouping Operator

- OQL also supports a grouping operator called `group by`
- To retrieve average GPA of majors in each department having >100 majors:

```
select deptname, avg_gpa:  
avg (select p.s.gpa from p in partition)  
from s in students  
group by deptname: s.majors_in.dname  
having count (partition) > 100
```

4. C++ Language Binding

- C++ language binding specifies how ODL constructs are mapped to C++ statements and include:
 - a C++ class library
 - a Data Manipulation Language (ODL/OML)
 - a set of constructs called *physical pragmas* (to allow programmers some control over the physical storage concerns)

Class Library

- The class library added to C++ for the ODMG standards uses the prefix `_d` for class declarations
- `d_Ref<T>` is defined for each database class `T`
- To utilize ODMG's collection types, various templates are defined, e.g., `d_Object<T>` specifies the operations to be inherited by all objects

Template Classes

- A template class is provided for each type of ODMG collections:

- `d_Set<T>`
 - `d_List<T>`
 - `d_Bag<t>`
 - `d_Varray<t>`
 - `d_Dictionary<T>`

- Thus a programmer can declare:

```
d_Set<d_Ref<Student>>
```

Data Types of Attributes

- The data types of ODMG database attributes are also available to the C++ programmers via the `_d` prefix, e.g., `d_Short`, `d_Long`, `d_Float`
- Certain structured literals are also available, e.g., `d_Date`, `d_Time`, `d_Intreval`

Specifying Relationships

- To specify relationships, the prefix Rel_ is used within the prefix of type names, e.g.,
`d_Rel_Ref<Dept, has_majors> majors_in;`
- The C++ binding also allows the creation of extents via using the library class
`d_Extent:`

`d_Extent<Person> All_Persons (CS101)`

21.5 Object Database Conceptual Design

- Object Database (ODB) vs Relational Database (RDB)
 - Relationships are handled differently
 - Inheritance is handled differently
 - Operations in OBD are expressed early on since they are a part of the class specification

Relationships: ODB vs RDB (1)

● Relationships in ODB:

- relationships are handled by reference attributes that include OIDs of related objects
- single and collection of references are allowed
- references for binary relationships can be expressed in single direction or both directions via inverse operator

Relationships: ODB vs RDB (2)

● Relationships in RDB:

- Relationships among tuples are specified by attributes with matching values (via *foreign keys*)
- Foreign keys are single-valued
- $M:N$ relationships must be presented via a separate relation (table)

Inheritance Relationship in ODB vs RDB

- Inheritance structures are built in ODB (and achieved via “:” and extends operators)
- RDB has no built-in support for inheritance relationships; there are several options for mapping inheritance relationships in an RDB (see Chapter 7)

Early Specification of Operations

- Another major difference between ODB and RDB is the specification of operations
 - ODB: operations specified during design (as part of class specification)
 - RDB: may be delayed until implementation

Mapping EER Schemas to ODB Schemas

- Mapping EER schemas into ODB schemas is relatively simple especially since ODB schemas provide support for inheritance relationships
- Once mapping has been completed, operations must be added to ODB schemas since EER schemas do not include an specification of operations

Mapping EER to ODB Schemas

Step 1

- Create an ODL class for each EER entity type or subclass
 - Multi-valued attributes are declared by sets, bags or lists constructors
 - Composite attributes are mapped into tuple constructors

Mapping EER to ODB Schemas

Step 2

- Add relationship properties or reference attributes for each binary relationship into the ODL classes participating in the relationship
 - Relationship cardinality: single-valued for 1:1 and N:1 directions; set-valued for 1:N and M:N directions
 - Relationship attributes: create via tuple constructors

Mapping EER to ODB Schemas

Step 3

- Add appropriate operations for each class
 - Operations are not available from the EER schemas; original requirements must be reviewed
 - Corresponding *constructor* and *destructor* operations must also be added

Mapping EER to ODB Schemas

Step 4

- Specify inheritance relationships via `extends` clause
 - An ODL class that corresponds to a sub-class in the EER schema inherits the types and methods of its super-class in the ODL schemas
 - Other attributes of a sub-class are added by following Steps 1-3

Mapping EER to ODB Schemas

Step 5

- Map weak entity types in the same way as regular entities
 - Weak entities that do not participate in any relationships may alternatively be presented as *composite multi-valued attribute* of the owner entity type

Mapping EER to ODB Schemas

Step 6

- Map categories (union types) to ODL
 - The process is not straightforward
 - May follow the same mapping used for EER-to-relational mapping:
 - Declare a class to represent the category
 - Define 1:1 relationships between the category and each of its super-classes

Mapping EER to ODB Schemas

Step 7

- Map n -ary relationships whose degree is greater than 2
 - Each relationship is mapped into a separate class with appropriate reference to each participating class

21.6 Summary

- Proposed standards for object databases presented
- Various constructs and built-in types of the ODMG model presented
- ODL and OQL languages were presented
- An overview of the C++ language binding was given
- Conceptual design of object-oriented database discussed

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 22

Object-Relational and Extended-Relational Systems



Overview of SQL and Its Object-Relational Features

- The SQL Standard and Its Components
- Object-Relational Support in SQL-99
- Some New Operations and Features in SQL

Evolution and Current Trends of Database Technology

The Informix Universal Server

- Extensible Data Types
- Support for User-Defined Routines
- Support for Inheritance
- Support for Indexing Extensions
- Support for External Data Sources
- Support for Data Blades Application Programming Interface

Object-Relational Features of Oracle 8

- Some Examples of Object-Relational Features of Oracle
- Managing Large Objects and Other Storage Features

Implementation and Related Issues for Extended Type Systems

The Nested Relational Model

Summary

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

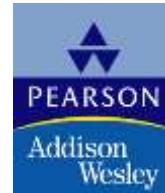
ELMASRI  NAVATHE

Chapter 23

Database Security

and

Authorization



Chapter Outline

1 Database Security and Authorization

- 1.1 Introduction to Database Security Issues
- 1.2 Types of Security
- 1.3 Database Security and DBA
- 1.4 Access Protection, User Accounts, and Database Audits

2 Discretionary Access Control Based on Granting Revoking Privileges

- 2.1 Types of Discretionary Privileges
- 2.2 Specifying Privileges Using Views
- 2.3 Revoking Privileges
- 2.4 Propagation of Privileges Using the GRANT OPTION
- 2.5 Specifying Limits on Propagation of Privileges

Chapter Outline(contd.)

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security

3.1 Comparing Discretionary Access Control and Mandatory Access Control

3.2 Role-Based Access Control

3.3 Access Control Policies for E-Commerce and the Web

4 Introduction to Statistical Database Security

Chapter Outline(contd.)

5 Introduction to Flow Control

5.1 Covert Channels

6 Encryption and Public Key Infrastructures

6.1 The Data and Advanced Encryption Standards

6.2 Public Key Encryption

6.3 Digital Signatures

1 Introduction to Database Security Issues

● Types of Security

- Legal and ethical issues
- Policy issues
- System-related issues
- The need to identify multiple security levels

Introduction to Database Security Issues (2)

Threats to databases

- Loss of integrity
- Loss of availability
- Loss of confidentiality

To protect databases against these types of threats four kinds of countermeasures can be implemented : *access control, inference control, flow control, and encryption.*

Introduction to Database Security Issues (3)

A DBMS typically includes a database security and authorization subsystem that is responsible for ensuring the security portions of a database against unauthorized access.

Two types of database security mechanisms:

- Discretionary security mechanisms
- Mandatory security mechanisms

Introduction to Database Security Issues (4)

The security mechanism of a DBMS must include provisions for restricting access to the database as a whole; this function is called **access control** and is handled by creating user accounts and passwords to control login process by the DBMS.

Introduction to Database Security Issues (5)

The security problem associated with databases is that of controlling the access to a **statistical database**, which is used to provide statistical information or summaries of values based on various criteria.

The countermeasures to **statistical database security** problem is called **inference control** measures.

Introduction to Database Security Issues (6)

Another security is that of **flow control**, which prevents information from flowing in such a way that it reaches unauthorized users.

Channels that are pathways for information to flow implicitly in ways that violate the security policy of an organization are called **covert channels**.

Introduction to Database Security Issues (7)

A final security issue is **data encryption**, which is used to protect sensitive data (such as credit card numbers) that is being transmitted via some type communication network.

The data is **encoded** using some coding algorithm. An unauthorized user who access encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher data.

1.2 Database Security and the DBA

The database administrator (DBA) is the central authority for managing a database system. The DBA's responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization. The DBA has a **DBA account** in the DBMS, sometimes called a **system or superuser account**, which provides powerful capabilities :

1.2 Database Security and the DBA

1. *Account creation*
2. *Privilege granting*
3. *Privilege revocation*
4. *Security level assignment*

The DBA is responsible for the overall security of the database system.

Action 1 is access control, whereas 2 and 3 are discretionary and 4 is used to control mandatory authorization.

1.3 Access Protection, User Accounts, and Database Audits

Whenever a person or group of persons need to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account number** and **password** for the user if there is a legitimate need to access the database.

The user must **log in** to the DBMS by entering account number and password whenever database access is needed.

1.3 Access Protection, User Accounts, and Database Audits(2)

The database system must also keep track of all operations on the database that are applied by a certain user throughout each **login session**.

To keep a record of all updates applied to the database and of the particular user who applied each update, we can modify **system log**, which includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash.

If any tampering with the database is suspected, a **database audit** is performed, which consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period.

A database log that is used mainly for security purposes is sometimes called an **audit trail**.

Discretionary Access Control Based on Granting and Revoking Privileges

The typical method of enforcing **discretionary access control** in a database system is based on the granting and revoking **privileges**.

2.1 Types of Discretionary Privileges

- The *account level*: At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- The *relation (or table level)*: At this level, the DBA can control the privilege to access each individual relation or view in the database.

2.1 Types of Discretionary Privileges(2)

The privileges at the **account level** apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such adding or removing attributes from relations; the DROP privilege, to delete relations or views; the MODIFY privilege, to insert, delete, or update tuples; and the SELECT privilege, to retrieve information from the database by using a SELECT query.

2.1 Types of Discretionary Privileges(3)

The second level of privileges applies to the **relation level**, whether they are base relations or virtual (view) relations.

The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the **access matrix model**, where the rows of a matrix M represents *subjects* (users, accounts, programs) and the columns represent *objects* (relations, records, columns, views, operations). Each position $M(i,j)$ in the matrix represents the types of privileges (read, write, update) that subject i holds on object j .

2.1 Types of Discretionary Privileges(4)

To control the granting and revoking of relation privileges, each relation R in a database is assigned and **owner account**, which is typically the account that was used when the relation was created in the first place. The owner of a relation is given *all* privileges on that relation. In SQL2, the DBA can assign and owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command. The owner account holder can pass privileges on any of the owned relation to other users by **granting** privileges to their accounts.

2.1 Types of Discretionary Privileges(5)

In SQL the following types of privileges can be granted on each individual relation R:

- SELECT (retrieval or read) privilege on R: Gives the account retrieval privilege. In SQL this gives the account the privilege to use the SELECT statement to retrieve tuples from R.
- MODIFY privileges on R: This gives the account the capability to modify tuples of R. In SQL this privilege is further divided into UPDATE, DELETE, and INSERT privileges to apply the corresponding SQL command to R. In addition, both the INSERT and UPDATE privileges can specify that only certain attributes can be updated by the account.

2.1 Types of Discretionary Privileges(6)

- REFERENCES privilege on R: This gives the account the capability to reference relation R when specifying integrity constraints. The privilege can also be restricted to specific attributes of R.

Notice that to create a view, the account must have SELECT privilege on *all relations* involved in the view definition.

2.2 Specifying Privileges Using Views

The mechanism of **views** is an important discretionary authorization mechanism in its own right.

For example, if the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that includes only those attributes and then grant SELECT on V to B. The same applies to limiting B to retrieving only certain tuples of R; a view V' can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

2.3 Revoking Privileges

In some cases it is desirable to grant a privilege to a user temporarily.

For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking** privileges is needed. In SQL, a REVOKE command is included for the purpose of canceling privileges.

2.4 Propagation of Privileges using the GRANT OPTION

Whenever the owner A of a relation R grants a privilege on R to another account B, privilege can be given to B *with* or *without* the GRANT OPTION. If the GRANT OPTION is given, this means that B can also grant that privilege on R to other accounts. Suppose that B is given the GRANT OPTION by A and that B then grants the privilege on R to a third account C, also with GRANT OPTION. In this way, privileges on R can **propagate** to other accounts without the knowledge of the owner of R. If the owner account A now revokes the privilege granted to B, all the privileges that B propagated based on that privilege should automatically be revoked by the system.

2.5 An Example

Suppose that the DBA creates four accounts --A1, A2, A3, and A4-- and wants only A1 to be able to create base relations; then the DBA must issue the following GRANT command in SQL:

`GRANT CREATETAB TO A1;`

In SQL2 the same effect can be accomplished by having the DBA issue a CREATE SCHEMA command as follows:

`CREATE SCHEMA EXAMPLE AUTHORIZATION A1;`

2.5 An Example(2)

User account A1 can create tables under the schema called EXAMPLE.

Suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT; A1 is then **owner** of these two relations and hence *all the relation privileges* on each of them.

Suppose that A1 wants to grant A2 the privilege to insert and delete tuples in both of these relations, but A1 does not want A2 to be able to propagate these privileges to additional accounts:

GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;

2.5 An Example(3)

EMPLOYEE

NAME	SSN	BDATE	ADDRESS	SEX	SALARY	DNO

DNUMBER	DNAME	MGRSSN

2.5 An Example(4)

Suppose that A1 wants to allow A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts.

A1 can issue the command:

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH  
GRANT OPTION;
```

A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

(Notice that A4 can not propagate the SELECT privilege because GRANT OPTION was not given to A4.)

2.5 An Example(5)

Suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 can issue:

REVOKE SELECT ON EMPLOYEE FROM A3;

(The DBMS must now automatically revoke the SELECT privilege on EMPLOYEE from A4, too, because A3 granted that privilege to A4 and A3 does not have the privilege any more.)

2.5 An Example(6)

Suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the NAME, BDATE, and ADDRESS attributes and only for the tuples with DNO=5.

A1 then create the view:

```
CREATE VIEW A3EMPLOYEE AS  
SELECT NAME, BDATE, ADDRESS  
FROM EMPLOYEE  
WHERE DNO = 5;
```

After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION;
```

2.5 An Example(7)

Finally, suppose that A1 wants to allow A4 to update only the SALARY attribute of EMPLOYEE;

A1 can issue:

GRANT UPDATE ON EMPLOYEE (SALARY) TO A4;

(The UPDATE or INSERT privilege can specify particular attributes that may be updated or inserted in a relation. Other privileges (SELECT, DELETE) are not attribute specific.)

2.6 Specifying Limits on Propagation of Privileges

Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and are not a part of SQL.

Limiting **horizontal propagation** to an integer number i means that an account B given the GRANT OPTION can grant the privilege to at most i other accounts.

Vertical propagation is more complicated; it limits the depth of the granting of privileges.

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security

The discretionary access control techniques of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems.

This is an all-or-nothing method: A user either has or does not have a certain privilege.

In many applications, and *additional security policy* is needed that classifies data and users based on security classes.

This approach as **mandatory access control**, would typically be *combined* with the discretionary access control mechanisms.

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(2)

Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest: $TS \geq S \geq C \geq U$

The commonly used model for multilevel security, known as the Bell-LaPadula model, classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications, T, S, C, or U: **clearance** (classification) of a subject S as **class(S)** and to the **classification** of an object O as **class(O)**.

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(3)

Two restrictions are enforced on data access based on the subject/object classifications:

1. A subject S is not allowed read access to an object O unless $\text{class}(S) \geq \text{class}(O)$. This is known as the **simple security property**.
2. A subject S is not allowed to write an object O unless $\text{class}(S) \leq \text{class}(O)$. This known as the **star property** (or * property).

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(4)

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a **classification attribute** C in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a **tuple classification** attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. Hence, a **multilevel relation** schema R with n attributes would be represented as

$R(A_1, C_1, A_2, C_2, \dots, A_n, C_n, TC)$

where each C_i represents the classification attribute associated with attribute A_i .

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(5)

The value of the TC attribute in each tuple t – which is the *highest* of all attribute classification values within t – provides a general classification for the tuple itself, whereas each C_i provides a finer security classification for each attribute value within the tuple.

The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular(single-level) relation.

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(6)

A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower-level classification through a process known as **filtering**.

In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the *apparent key*. This leads to the concept of **polyinstantiation** where several tuples can have the same apparent key value but have different attribute values for users at different classification levels.

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(7)

In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the *same* security classification within each individual tuple.

In addition, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key. This constraint ensures that a user can see the key if the user is permitted to see any part of the tuple at all.

3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(8)

Other integrity rules, called **null integrity** and **interinstance integrity**, informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

3.1 Comparing Discretionary Access Control and Mandatory Access Control

- Discretionary Access Control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains.
- The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs.

3.1 Comparing Discretionary Access Control and Mandatory Access Control(2)

- By contrast, mandatory policies ensure a high degree of protection in a way, they prevent any illegal flow of information.
- Mandatory policies have the drawback of being too rigid and they are only applicable in limited environments.
- In many practical situations, discretionary policies are preferred because they offer a better trade-off between security and applicability.

3.2 Role-Based Access Control

Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise-wide systems. Its basic notion is that permissions are associated with roles, and users are assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands. The GRANT and REVOKE commands discussed under DAC can then be used to assign and revoke privileges from roles.

3.2 Role-Based Access Control(2)

- RBAC appears to be a viable alternative to traditional discretionary and mandatory access controls; it ensures that only authorized users are given access to certain data or resources.
- Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.
- Role hierarchy in RBAC is a natural way of organizing roles to reflect the organization's lines of authority and responsibility.

3.2 Role-Based Access Control(3)

- Another important consideration in RBAC systems is the possible temporal constraints that may exist on roles, such as time and duration of role activations, and timed triggering of a role by an activation of another role.
- Using an RBAC model is highly desirable goal for addressing the key security requirements of Web-based applications.

In contrast, discretionary access control (DAC) and mandatory access control (MAC) models lack capabilities needed to support the security requirements emerging enterprises and Web-based applications.

3.3 Access Control Policies for E-Commerce and the Web

- **E-Commerce** environments require elaborate policies that go beyond traditional DBMSs.
 - In an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience.
 - The access control mechanism should be flexible enough to support a wide spectrum of heterogeneous protection objects.
- A related requirement is the support for content-based access-control.

3.3 Access Control Policies for E-Commerce and the Web(2)

- Another requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications.
 - A possible solution, to better take into account user profiles in the formulation of access control policies, is to support the notion of credentials. A **credential** is a set of properties concerning a user that are relevant for security purposes (for example, age, position within an organization).
 - It is believed that the XML language can play a key role in access control for e-commerce applications.

4 Introduction to Statistical Database Security

- Statistical databases are used mainly to produce statistics on various populations.
- The database may contain confidential data on individuals, which should be protected from user access.
- Users are permitted to retrieve statistical information on the populations, such as averages, sums, counts, maximums, minimums, and standard deviations.

4 Introduction to Statistical Database Security(2)

A **population** is a set of tuples of a relation (table) that satisfy some selection condition.

- Statistical queries involve applying statistical functions to a population of tuples.

4 Introduction to Statistical Database Security(3)

For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person. Statistical database security techniques must prohibit the retrieval of individual data.

This can be achieved by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called **statistical queries**.

4 Introduction to Statistical Database Security(4)

- It is DBMS's responsibility to ensure confidentiality of information about individuals, while still providing useful statistical summaries of data about those individuals to users.
- Provision of **privacy protection** of users in a statistical database is paramount.
- In some cases it is possible to **infer** the values of individual tuples from a sequence statistical queries. This is particularly true when the conditions result in a population consisting of a small number of tuples.

5 Introduction to Flow Control

- Flow control regulates the distribution or flow of information among accessible objects. A flow between object X and object Y occurs when a program reads values from X and writes values into Y.
- Flow controls check that information contained in some objects does not flow explicitly or implicitly into less protected objects.
- A flow policy specifies the channels along which information is allowed to move. The simplest flow policy specifies just two classes of information: confidential (C) and nonconfidential (N), and allows all flows except those from class C to class N.

5.1 Covert Channels

A covert channel allows a transfer of information that violates the security or the policy.

- A **covert channel** allows information to pass from a higher classification level to a lower classification level through improper means.

5.1 Covert Channels(2)

- Covert channels can be classified into two broad categories:
 - **storage channels** do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.
 - in a **timing channel** the information is conveyed by the timing of events or processes.

Some security experts believe that one way to avoid covert channels is for programmers to not actually gain access to sensitive data that a program is supposed to process after the program has been put into operation.

6 Encryption and Public Key Infrastructures

- Encryption is a means of maintaining secure data in an insecure environment.
- Encryption consists of applying an **encryption algorithm** to data using some prespecified **encryption key**.
 - the resulting data has to be **decrypted** using a **decryption key** to recover the original data.

6.1 The Data and Advanced Encryption Standards

- The Data Encryption Standard (DES) is a system developed by the U.S. government for use by the general public. It has been widely accepted as a cryptographic standard both in the United States and abroad.
- DES can provide end-to-end encryption on the channel between the sender A and receiver B.

6.1 The Data and Advanced Encryption Standards(2)

- DES algorithm is a careful and complex combination of two of the fundamental building blocks of encryption: substitution and permutation (transposition).
 - The algorithm derives its strength from repeated application of these two techniques for a total of 16 cycles.
 - Plaintext (the original form of the message) is encrypted as blocks of 64 bits.
- After questioning the adequacy of DES, the National Institute of Standards (NIST) introduced the Advanced Encryption Standards (AES).
 - This algorithm has a block size of 128 bits and thus takes longer time to crack.

6.2 Public Key Encryption

In 1976 Diffie and Hellman proposed a new kind of cryptosystem, which they called **public key encryption**.

- Public key algorithms are based on mathematical functions rather than operations on bit patterns.
 - They also involve the use of two separate keys, in contrast to conventional encryption, which uses only one key.
 - The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication.

6.2 Public Key Encryption(2)

- The two keys used for public key encryption are referred to as the **public key** and the **private key**.
 - the private key is kept secret, but it is referred to as *private key* rather than a *secret key* (the key used in conventional encryption) to avoid confusion with conventional encryption.

6.2 Public Key Encryption(3)

A public key encryption scheme, or infrastructure, has six ingredients:

1. *Plaintext* : This is the data or readable message that is fed into the algorithm as input.
2. *Encryption algorithm* : The encryption algorithm performs various transformations on the plaintext.
3. and
4. *Public and private keys* : These are pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exec transformations performed by the encryption algorithm depend on the public or private key that is provided as input.

6.2 Public Key Encryption(4)

5. *Ciphertext* : This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
6. *Decryption algorithm* : This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

6.2 Public Key Encryption(5)

Public key is made for public and private key is known only by owner.

A general-purpose public key cryptographic algorithm relies on one key for encryption and a different but related one for decryption. The essential steps are as follows:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private.

6.2 Public Key Encryption(6)

3. If a sender wishes to send a private message to a receiver, the sender encrypts the message using the receiver's public key.
4. When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.

6.2 Public Key Encryption(7)

The RSA Public Key Encryption algorithm, one of the first public key schemes was introduced in 1978 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and is named after them as the RSA scheme.

- The RSA encryption algorithm incorporates results from number theory, combined with the difficulty of determining the prime factors of a target.
- The RSA algorithm also operates with modular arithmetic – mod n.

6.2 Public Key Encryption(8)

- Two keys, d and e , are used for decryption and encryption.
 - An important property is that d and e can be interchanged.
 - n is chosen as a large integer that is a product of two large distinct prime numbers, a and b .
 - The encryption key e is a randomly chosen number between 1 and n that is relatively prime to $(a-1) \times (b-1)$.
 - The plaintext block P is encrypted as $P^e \text{ mod } n$.
 - Because the exponentiation is performed $\text{mod } n$, factoring P^e to uncover the encrypted plaintext is difficult.
 - However, the decryption key d is carefully chosen so that $(P^e)^d \text{ mod } n = P$.
 - The decryption key d can be computed from the condition that $d \times e = 1 \text{ mod } ((a-1)(b-1))$.
 - Thus, the legitimate receiver who knows d simply computes $(P^e)^d \text{ mod } n = P$ and recovers P without having to factor P^e .

6.3 Digital Signatures

A digital signature is an example of using encryption techniques to provide authentication services in e-commerce applications.

- A **digital signature** is a means of associating a mark unique to an individual with a body of text.
 - The mark should be unforgettable, meaning that others should be able to check that the signature does come from the originator.
- A **digital signature** consists of a string of symbols.
 - Signature must be different for each use. This can be achieved by making each digital signature a function of the message that it is signing, together with a time stamp.
 - Public key techniques are the means creating digital signatures.

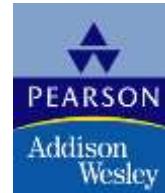
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 24

Enhanced Data Models for Advanced Applications



Copyright © 2004 Pearson Education, Inc.

Active Database Concepts and Triggers

- Generalized Model for Active Databases and Oracle Triggers
- Design and Implementation Issues for Active Databases
- Examples of Statement-Level Active Rules in STARBURST
- Potential Applications for Active Databases
- Triggers in SQL-99

Temporal Database Concepts

- Time Representation, Calendars, and Time Dimensions
- Incorporating Time in Relational Databases Using Tuple Versioning
- Incorporating Time in Object-Oriented Databases Using Attribute Versioning
- Temporal Querying Constructs and the TSQL2 Language
- Time Series Data

Multimedia Databases

- Introduction to Spatial Database Concepts
- Introduction to Multimedia Database Concepts

Introduction to Deductive Databases

- Overview of Deductive Databases
- Prolog/Datalog Notation
- Datalog Notation
- Clausal Form and Horn Clauses
- Interpretation of Rules
- Datalog Programs and Their Safety
- Use the Relational Operations
- Evaluation of Nonrecursive Datalog Queries

Summary

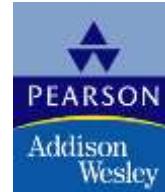
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 25

Distributed Databases and Client–Server Architectures



Distributed Database Concepts

- Parallel Versus Distributed Technology
- Advantages of Distributed Databases
- Additional Functions of Distributed Databases

Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design

- Data Fragmentation
- Data Replication and Allocation
- Example of Fragmentation, Allocation, and Replication

Types of Distributed Database Systems

Query Processing in Distributed Databases

- Data Transfer Costs of Distributed Query Processing
- Distributed Query Processing Using Semijoin
- Query and Update Decomposition

Overview of Concurrency Control and Recovery in Distributed Databases

- Distributed Concurrency Control Based on a Distinguished Copy of a Data Item
- Distributed Concurrency Control Based on Voting
- Distributed Recovery

An Overview of 3-Tier Client-Server Architecture

Distributed Databases in Oracle

Summary

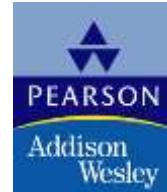
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 26

XML and Internet Databases



Chapter Outline

- Introduction
- Structured, Semi structured, and Unstructured Data.
- XML Hierarchical (Tree) Data Model.
- XML Documents, DTD, and XML Schema.
- XML Documents and Databases.
- XML Querying.
 - Xpath
 - XQuery

Introduction

- Although HTML is widely used for formatting and structuring Web *documents*, it is not suitable for specifying *structured data* that is extracted from databases.
- A new language—namely XML (eXtended Markup Language) has emerged as the standard for structuring and exchanging data over the Web. XML can be used to provide more information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen.
- The formatting aspects are specified separately—for example, by using a formatting language such as XSL (eXtended Stylesheet Language).

Structured, Semi Structured and Unstructured Data.

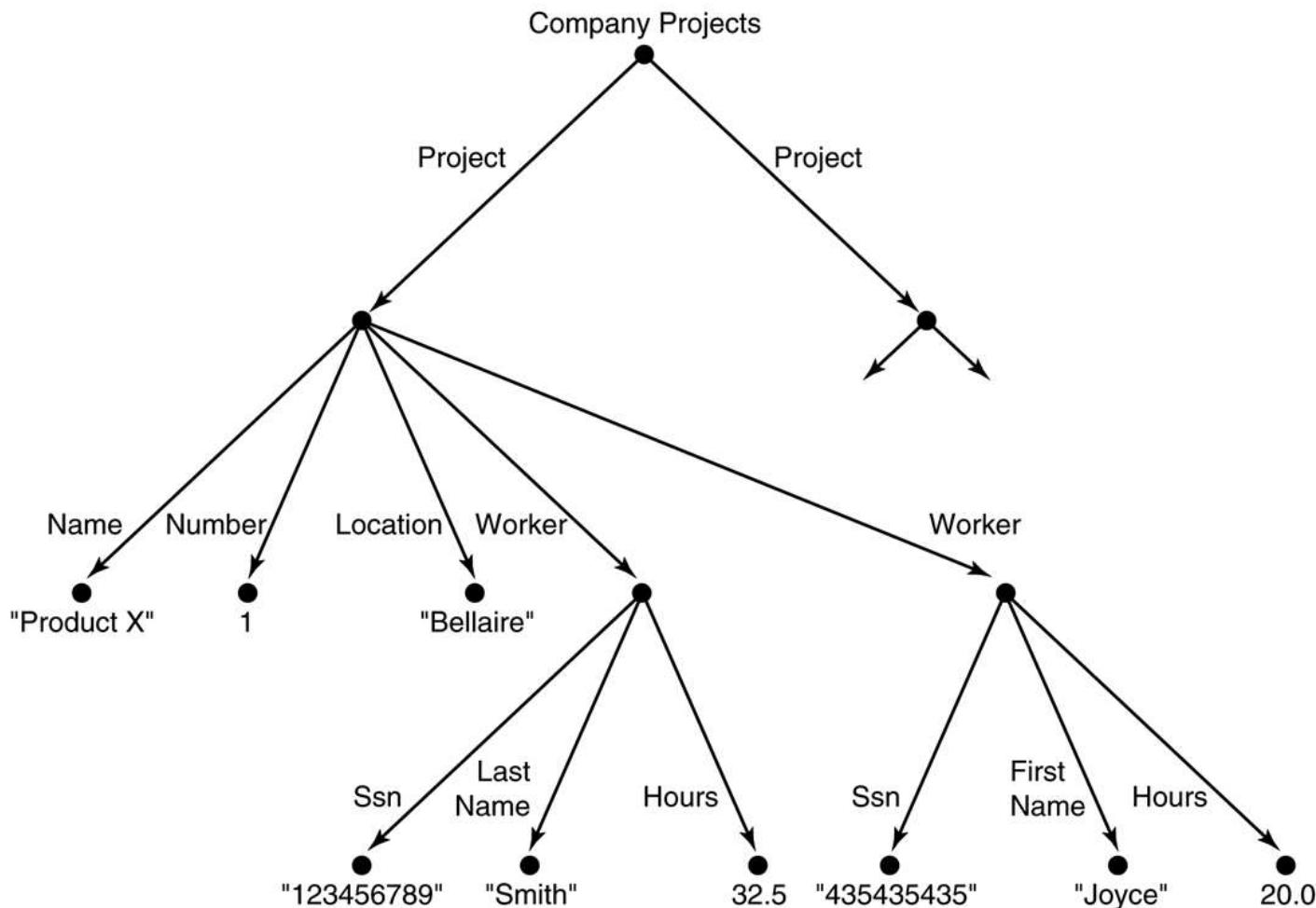
- Information stored in databases is known as **structured data** because it is represented in a strict format. The DBMS then checks to ensure that all data follows the structures and constraints specified in the schema.
- In some applications, data is collected in an ad-hoc manner before it is known how it will be stored and managed. This data may have a certain structure, but not all the information collected will have identical structure. This type of data is known as **semi-structured data**.
 - In semi-structured data, the schema information is *mixed in* with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as **self-describing data**.
- A third category is known as **unstructured data**, because there is very limited indication of the type of data. A typical example would be a text document that contains information embedded within it. Web pages in HTML that contain some data are considered as unstructured data.

Structured, Semi Structured and Unstructured Data (cont.)

- Semi-structured data may be displayed as a directed graph, as shown.
- The **labels** or **tags** on the directed edges represent the schema names—the names of attributes, object types (or entity types or classes), and relationships.
- The internal nodes represent individual objects or composite attributes.
- The leaf nodes represent actual data values of simple (atomic) attributes.

FIGURE 26.1

Representing semistructured data as a graph.



XML Hierarchical (Tree) Data Model

FIGURE 26.3
A complex XML
element called
`<projects>`.

```
<?xml version="1.0" standalone="yes"?>
<projects>

  <project>
    <Name>ProductX</Name>
    <Number>1</Number>
    <Location>Bellaire</Location>
    <DeptNo>5</DeptNo>
    <Worker>
      <SSN>123456789</SSN>
      <LastName>Smith</LastName>
      <hours>32.5</hours>
    </Worker>
    <Worker>
      <SSN>453453453</SSN>
      <FirstName>Joyce</FirstName>
      <hours>20.0</hours>
    </Worker>
  </project>
  <project>
    <Name>ProductY</Name>
    <Number>2</Number>
    <Location>Sugarland</Location>
    <DeptNo>5</DeptNo>
    <Worker>
      <SSN>123456789</SSN>
      <hours>7.5</hours>
    </Worker>
    <Worker>
      <SSN>453453453</SSN>
      <hours>20.0</hours>
    </Worker>
    <Worker>
      <SSN>333445555</SSN>
      <hours>10.0</hours>
    </Worker>
  </project>
  ...
</projects>
```

XML Hierarchical (Tree) Data Model (cont.)

- The basic object in XML is the **XML document**. There are two main structuring concepts that are used to construct an XML document: **elements** and **attributes**. Attributes in XML provide additional information that describe elements.
- As in HTML, elements are identified in a document by their **start tag** and **end tag**. The tag names are enclosed between angled brackets `<...>`, and end tags are further identified by a backslash `</...>`. **Complex elements** are constructed from other elements hierarchically, whereas **simple elements** contain data values.
- It is straightforward to see the correspondence between the XML textual representation and the tree structure. In the tree representation, internal nodes represent complex elements, whereas leaf nodes represent simple elements. That is why the XML model is called a **tree model** or a **hierarchical model**.

XML Hierarchical (Tree) Data Model (cont.)

It is possible to characterize three main types of XML documents:

1. *Data-centric XML documents:*

These documents have many small data items that follow a specific structure, and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them or display them over the Web.

2. *Document-centric XML documents:*

These are documents with large amounts of text, such as news articles or books. There is little or no structured data elements in these documents.

3. *Hybrid XML documents:*

These documents may have parts that contains structured data and other parts that are predominantly textual or unstructured.

XML Documents, DTD, and XML Schema.

Well-Formed

- It must start with an XML declaration to indicate the version of XML being used—as well as any other relevant attributes.
- It must follow the syntactic guidelines of the tree model. This means that there should be a ***single root element***, and every element must include a matching pair of start tag and end tag within the start and end tags ***of the parent element***.
- A well-formed XML document is **syntactically correct**. This allows it to be processed by generic processors that traverse the document and create an internal tree representation.
 - **DOM (Document Object Model)** - Allows programs to manipulate the resulting tree representation corresponding to a well-formed XML document. The whole document must be parsed beforehand when using dom.
 - **SAX** - Allows processing of XML documents on the fly by notifying the processing program whenever a start or end tag is encountered.

Valid

- A stronger criterion is for an XML document to be **valid**. In this case, the document must be well-formed, and in addition the element names used in the start and end tag pairs must follow the structure specified in a separate XML **DTD (Document Type Definition)** file or XML schema file.

XML Documents, DTD, and XML Schema (cont.)

FIGURE 26.4 An XML DTD file called projects.

```
<!DOCTYPE projects [
  <!ELEMENT projects (project+)>
  <!ELEMENT project (Name, Number, Location, DeptNo?, Workers)>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Number (#PCDATA)>
  <!ELEMENT Location (#PCDATA)>
  <!ELEMENT DeptNo (#PCDATA)>
  <!ELEMENT Workers (Worker*)>
  <!ELEMENT Worker (SSN, LastName?, FirstName?, hours)>
  <!ELEMENT SSN (#PCDATA)>
  <!ELEMENT LastName (#PCDATA)>
  <!ELEMENT FirstName (#PCDATA)>
  <!ELEMENT hours (#PCDATA)>
]>
```

XML Documents, DTD, and XML Schema (cont.)

XML DTD Notation

- A * following the element name means that the element can be repeated zero or more times in the document. This can be called an optional multivalued (repeating) element.
- A + following the element name means that the element can be repeated one or more times in the document. This can be called a required multivalued (repeating) element.
- A ? following the element name means that the element can be repeated zero or one times. This can be called an optional single-valued (non-repeating) element.
- An element appearing without any of the preceding three symbols must appear exactly once in the document. This can be called an required single-valued (non-repeating) element.
- The type of the element is specified via parentheses following the element. If the parentheses include names of other elements, these would be the *children* of the element in the tree structure. If the parentheses include the keyword #PCDATA or one of the other data types available in XML DTD, the element is a leaf node. PCDATA stands for *parsed character data*, which is roughly similar to a string data type.
- Parentheses can be nested when specifying elements.
- A bar symbol ($e1 | e2$) specifies that either $e1$ or $e2$ can appear in the document.

XML Documents, DTD, and XML Schema (cont.)

Limitations of XML DTD

- First, the data types in DTD are not very general.
- DTD has its own special syntax and so it requires specialized processors. It would be advantageous to specify XML schema documents using the syntax rules of XML itself so that the same processors for XML documents can process XML schema descriptions.
- Third, all DTD elements are always forced to follow the specified ordering the document so unordered elements are not permitted.

XML Documents, DTD, and XML Schema (cont.)

FIGURE 26.5 An XML schema file called company.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">Company Schema (Element Approach) -
            Prepared by Babak Hojabri</xsd:documentation>
    </xsd:annotation>
    <xsd:element name="company">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="department" type="Department" minOccurs="0"
                    maxOccurs="unbounded" />
                <xsd:element name="employee" type="Employee" minOccurs="0"
                    maxOccurs="unbounded">
                    <xsd:unique name="dependentNameUnique">
                        <xsd:selector xpath="employeeDependent" />
                        <xsd:field xpath="dependentName" />
                    </xsd:unique>
                </xsd:element>
                <xsd:element name="project" type="Project" minOccurs="0"
                    maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

FIGURE 26.5 (continued)

An XML schema file called company.

```
<xsd:unique name="departmentNameUnique">
    <xsd:selector xpath="department" />
    <xsd:field xpath="departmentName" />
</xsd:unique>
<xsd:unique name=" projectNameUnique">
    <xsd:selector xpath="project" />
    <xsd:field xpath=" projectName" />
</xsd:unique>
<xsd:key name=" projectNumberKey" >
    <xsd:selector xpath="project" />
    <xsd:field xpath=" projectNumber" />
</xsd:key>
<xsd:key name=" departmentNumberKey" >
    <xsd:selector xpath="department" />
    <xsd:field xpath=" departmentNumber" />
</xsd:key>
<xsd:key name=" employeeSSNKey" >
    <xsd:selector xpath="employee" />
    <xsd:field xpath=" employeeSSN" />
</xsd:key>
<xsd:keyref name=" departmentManagerSSNKeyRef" refer=" employeeSSNKey" >
    <xsd:selector xpath="department" />
    <xsd:field xpath=" departmentManagerSSN" />
</xsd:keyref>
<xsd:keyref name=" employeeDepartmentNumberKeyRef" refer=" departmentNumberKey" >
    <xsd:selector xpath="employee" />
    <xsd:field xpath=" employeeDepartmentNumber" />
</xsd:keyref>
<xsd:keyref name=" employeeSupervisorSSNKeyRef" refer=" employeeSSNKey" >
    <xsd:selector xpath="employee" />
    <xsd:field xpath=" employeeSupervisorSSN" />
</xsd:keyref>
<xsd:keyref name=" projectDepartmentNumberKeyRef" refer=" departmentNumberKey" >
    <xsd:selector xpath="project" />
    <xsd:field xpath=" projectDepartmentNumber" />
</xsd:keyref>
<xsd:keyref name=" projectWorkerSSNKeyRef" refer=" employeeSSNKey" >
    <xsd:selector xpath="project/projectWorker" />
    <xsd:field xpath=" SSN" />
</xsd:keyref>
<xsd:keyref name=" employeeWorksOnProjectNumberKeyRef" refer=" projectNumberKey" >
    <xsd:selector xpath="employee/employeeWorksOn" />
    <xsd:field xpath=" projectNumber" />
</xsd:keyref>
</xsd:element>
```

FIGURE 26.5 (continued)

An XML schema file called company.

```
<xsd:complexType name="Department">
  <xsd:sequence>
    <xsd:element name="departmentName" type="xsd:string" />
    <xsd:element name="departmentNumber" type="xsd:string" />
    <xsd:element name="departmentManagerSSN" type="xsd:string" />
    <xsd:element name="departmentManagerStartDate" type="xsd:date" />
    <xsd:element name="departmentLocation" type="xsd:string"
      minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Employee">
  <xsd:sequence>
    <xsd:element name="employeeName" type="Name" />
    <xsd:element name="employeeSSN" type="xsd:string" />
    <xsd:element name="employeeSex" type="xsd:string" />
    <xsd:element name="employeeSalary" type="xsd:unsignedInt" />
    <xsd:element name="employeeBirthDate" type="xsd:date" />
    <xsd:element name="employeeDepartmentNumber" type="xsd:string" />
    <xsd:element name="employeeSupervisorSSN" type="xsd:string" />
    <xsd:element name="employeeAddress" type="Address" />
    <xsd:element name="employeeWorksOn" type="WorksOn" minOccurs="1"
      maxOccurs="unbounded" />
    <xsd:element name="employeeDependent" type="Dependent" minOccurs="0"
      maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Project">
  <xsd:sequence>
    <xsd:element name="projectName" type="xsd:string" />
    <xsd:element name="projectNumber" type="xsd:string" />
    <xsd:element name="projectLocation" type="xsd:string" />
    <xsd:element name="projectDepartmentNumber" type="xsd:string" />
    <xsd:element name="projectWorker" type="Worker" minOccurs="1"
      maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Dependent">
  <xsd:sequence>
    <xsd:element name="dependentName" type="xsd:string" />
    <xsd:element name="dependentSex" type="xsd:string" />
    <xsd:element name="dependentBirthDate" type="xsd:date" />
    <xsd:element name="dependentRelationship" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="number" type="xsd:string" />
    <xsd:element name="street" type="xsd:string" />
    <xsd:element name="city" type="xsd:string" />
    <xsd:element name="state" type="xsd:string" />
  </xsd:sequence>
```

FIGURE 26.5 (continued)

An XML schema file called company.

```
</xsd:complexType>
<xsd:complexType name="Name">
  <xsd:sequence>
    <xsd:element name="firstName" type="xsd:string" />
    <xsd:element name="middleName" type="xsd:string" />
    <xsd:element name="lastName" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Worker">
  <xsd:sequence>
    <xsd:element name="SSN" type="xsd:string" />
    <xsd:element name="hours" type="xsd:float" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="WorksOn">
  <xsd:sequence>
    <xsd:element name="projectNumber" type="xsd:string" />
    <xsd:element name="hours" type="xsd:float" />
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

XML Documents, DTD, and XML Schema (cont.)

XML Schema

- ***Schema Descriptions and XML Namespaces:***

It is necessary to identify the specific set of XML schema language elements (tags) by a file stored at a Web site location. The second line in our example specifies the file used in this example, which is:

"<http://www.w3.org/2001/XMLSchema>".

Each such definition is called an **XML namespace**.

The file name is assigned to the variable xsd using the attribute xmlns (XML namespace), and this variable is used as a prefix to all XML schema tags.

- ***Annotations, documentation, and language used:***

The xsd:annotation and xsd:documentation are used for providing comments and other descriptions in the XML document. The attribute XML:lang of the xsd:documentation element specifies the language being used. Eg. "en"

- ***Elements and types:***

We specify the *root element* of our XML schema. In XML schema, the name attribute of the xsd:element tag specifies the element name, which is called company for the root element in our example. The structure of the company root element is a xsd:complexType.

XML Documents, DTD, and XML Schema (cont.)

XML Schema

- ***First-level elements in the company database:***

These elements are named **employee**, **department**, and **project**, and each is specified in an xsd:element tag. If a tag has only attributes and no further sub-elements or data within it, it can be ended with the back slash symbol (/>) and termed **Empty Element**.

- ***Specifying element type and minimum and maximum occurrences:***

If we specify a type attribute in an xsd:element, this means that the structure of the element will be described separately, typically using the xsd:complexType element. The minOccurs and maxOccurs tags are used for specifying lower and upper bounds on the number of occurrences of an element. The default is exactly one occurrence.

- ***Specifying Keys:***

For specifying **primary keys**, the tag xsd:key is used.

For specifying **foreign keys**, the tag xsd:keyref is used. When specifying a foreign key, the attribute refer of the xsd:keyref tag specifies the referenced primary key whereas the tags xsd:selector and xsd:field specify the referencing element type and foreign key.

XML Documents, DTD, and XML Schema (cont.)

XML Schema

- ***Specifying the structures of complex elements via complex types:***

Complex elements in our example are **Department**, **Employee**, **Project**, and **Dependent**, which use the tag **xsd:complexType**. We specify each of these as a sequence of subelements corresponding to the database attributes of each entity type by using the **xsd:sequence** and **xsd:element** tags of XML schema. Each element is given a name and type via the attributes **name** and **type** of **xsd:element**.

We can also specify **minOccurs** and **maxOccurs** attributes if we need to change the default of exactly one occurrence. For (optional) database attributes where null is allowed, we need to specify **minOccurs = 0**, whereas for multivalued database attributes we need to specify **maxOccurs = “unbounded”** on the corresponding element.
- ***Composite (compound) attributes:***

Composite attributes from ER Schema are also specified as complex types in the XML schema, as illustrated by the **Address**, **Name**, **Worker**, and **WorkesOn** complex types. These could have been directly embedded within their parent elements.

XML Documents and Databases.

Approaches to Storing XML Documents

- ***Using a dbms to store the documents as text:***

We can use a relational or object dbms to store whole XML documents as text fields within the dbms records or objects. This approach can be used if the dbms has a special module for document processing, and would work for storing schemaless and document-centric XML documents.

- ***Using a dbms to store the document contents as data elements:***

This approach would work for storing a collection of documents that follow a specific XML DTD or XML schema. Since all the documents have the same structure, we can design a relational (or object) database to store the leaf-level data elements within the XML documents.

- ***Designing a specialized system for storing native XML data:***

A new type of database system based on the hierarchical (tree) model would be designed and implemented. The system would include specialized indexing and querying techniques, and would work for all types of XML documents.

- ***Creating or publishing customized XML documents from pre-existing relational databases:***

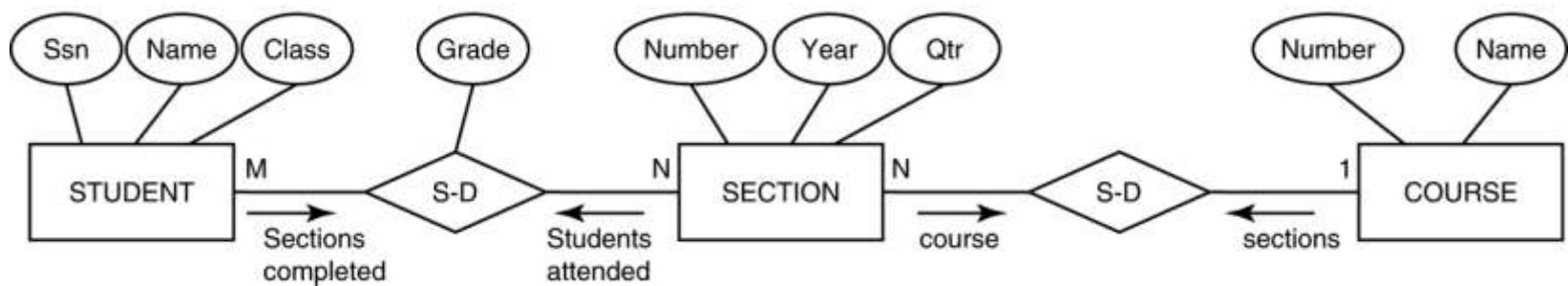
Because there are enormous amounts of data already stored in relational databases, parts of these data may need to be formatted as documents for exchanging or displaying over the Web.

XML Documents, DTD, and XML Schema (cont.)

Extracting XML Documents from Relational Databases.

Suppose that an application needs to extract XML documents for student, course, and grade information from the university database. The data needed for these documents is contained in the database attributes of the entity types **course, section, and student** as shown below (part of the main ER), and the relationships s-s and c-s between them.

FIGURE 26.7
Subset of the UNIVERSITY database schema needed for
XML document extraction.



XML Documents, DTD, and XML Schema (cont.)

Extracting XML Documents from Relational Databases.

One of the possible hierarchies that can be extracted from the database subset could choose **COURSE** as the root.

FIGURE 26.8
Hierarchical (tree)
view with COURSE
as the root.

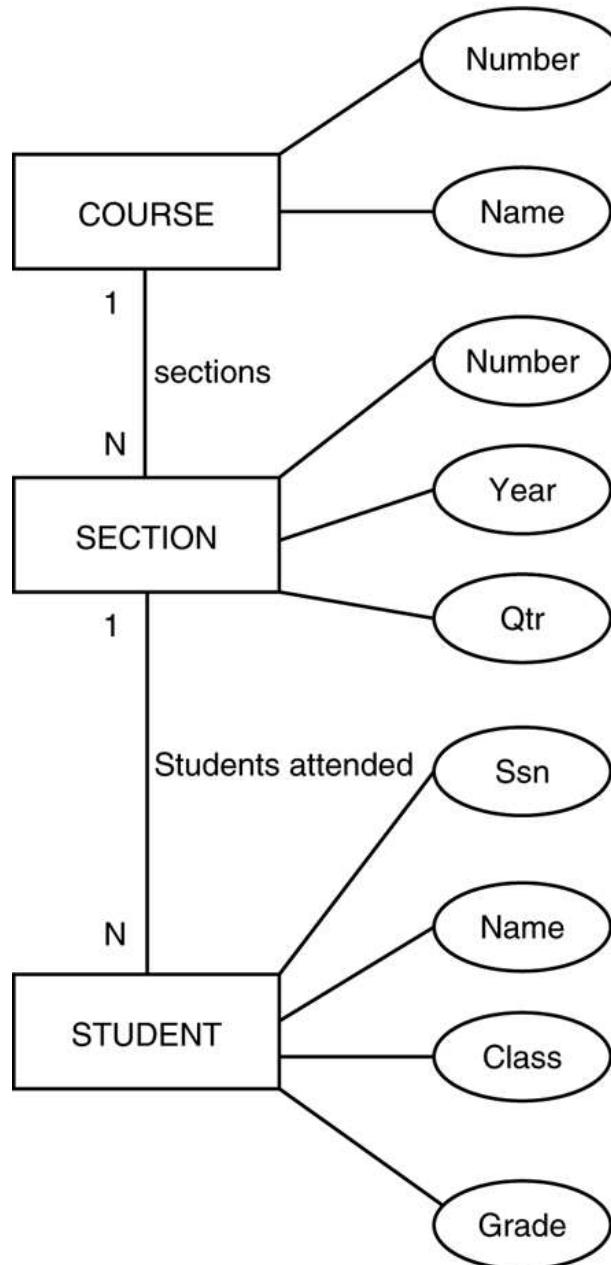


FIGURE 26.9 XML schema document with COURSE as the root.

```
<xsd:element name="root">
<xsd:sequence>
<xsd:element name="course" minOccurs="0" maxOccurs="unbounded">
    <xsd:sequence>
        <xsd:element name="cname" type="xsd:string" />
        <xsd:element name="cnumber" type="xsd:unsignedInt" />
        <xsd:element name="section" minOccurs="0" maxOccurs="unbounded">
            <xsd:sequence>
                <xsd:element name="secnumber" type="xsd:unsignedInt" />
                <xsd:element name="year" type="xsd:string" />
                <xsd:element name="quarter" type="xsd:string" />
                <xsd:element name="student" minOccurs="0" maxOccurs="unbounded">
                    <xsd:sequence>
                        <xsd:element name="ssn" type="xsd:string" />
                        <xsd:element name="sname" type="xsd:string" />
                        <xsd:element name="class" type="xsd:string" />
                        <xsd:element name="grade" type="xsd:string" />
                    </xsd:sequence>
                </xsd:element>
            </xsd:sequence>
        </xsd:element>
    </xsd:sequence>
</xsd:element>
</xsd:sequence>
</xsd:element>
```

XML Documents, DTD, and XML Schema (cont.)

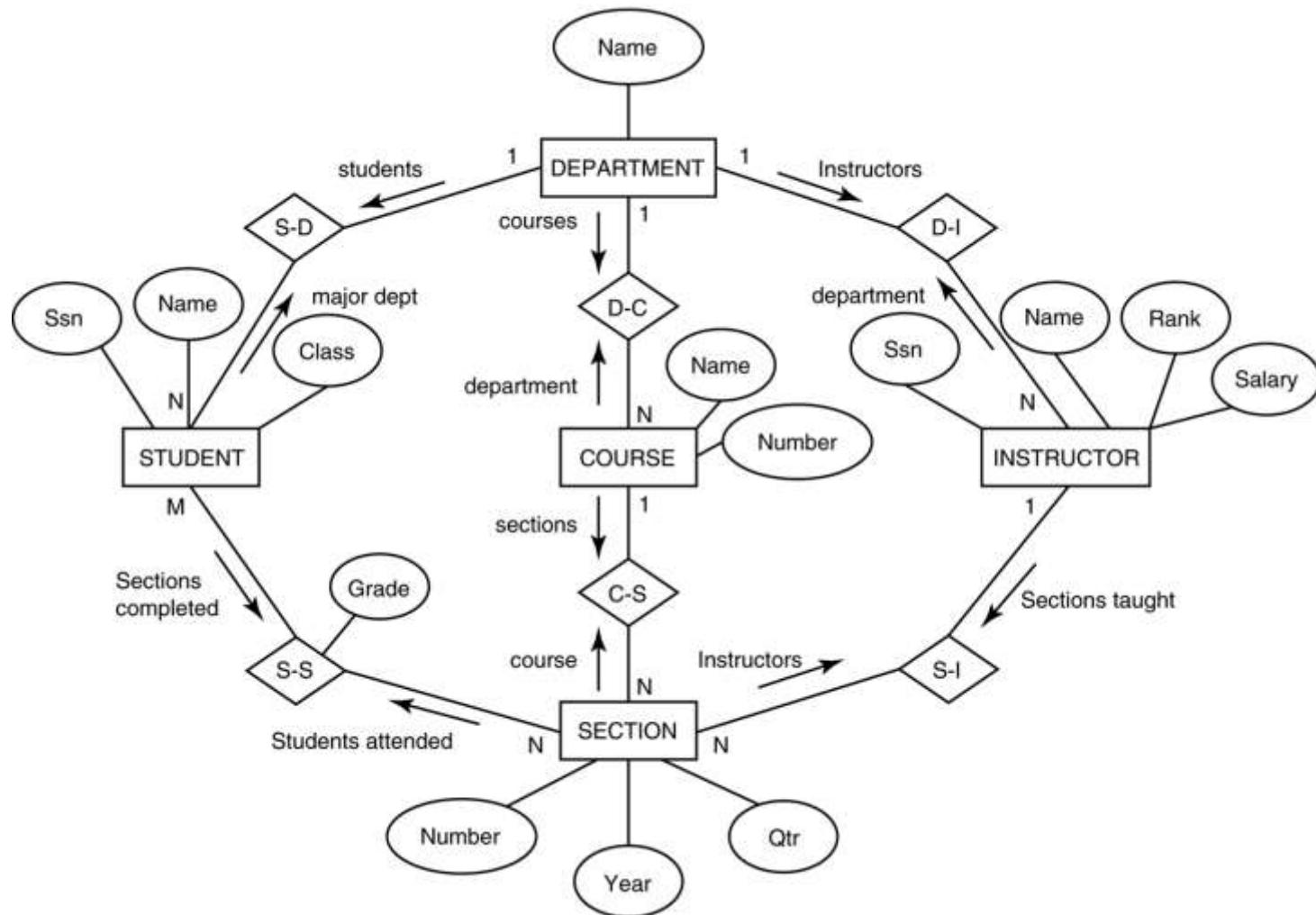
Breaking Cycles To Convert Graphs into Trees

It is possible to have a more complex subset with one or more cycles, indicating multiple relationships among the entities.

Suppose that we need the information in all the entity types and relationships in figure below for a particular XML document, with **student** as the root element.

FIGURE 26.6

An ER schema diagram for a simplified UNIVERSITY database.

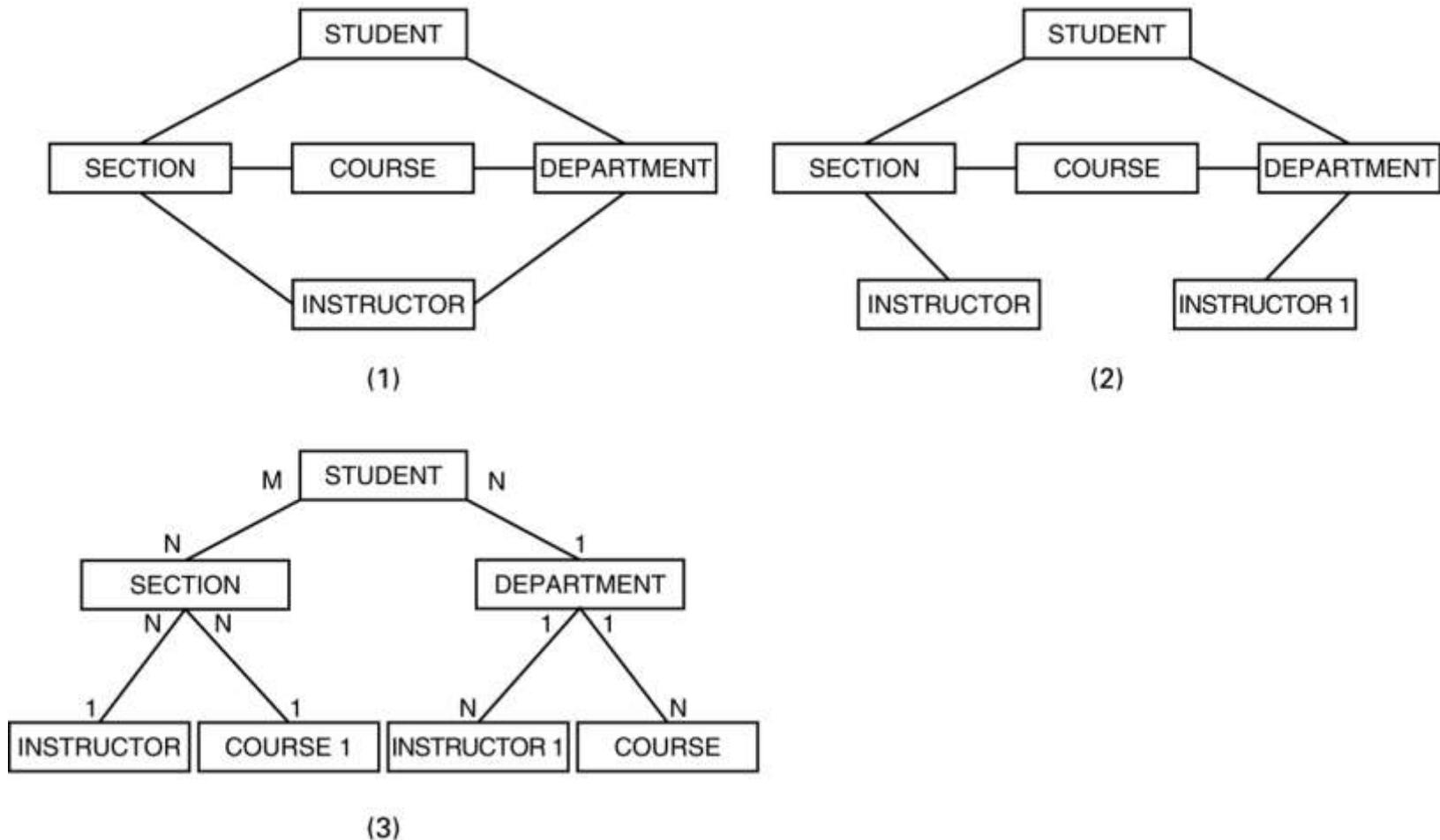


Breaking Cycles To convert Graphs into Trees

One way to break the cycles is to replicate the entity types involved in cycles.

- First, we replicate INSTRUCTOR as shown in part (2) of Figure, calling the replica to the right INSTRUCTOR1. The INSTRUCTOR replica on the left represents the relationship between instructors and the sections they teach, whereas the INSTRUCTOR1 replica on the right represents the relationship between instructors and the department each works in.
- We still have the cycle involving COURSE, so we can replicate COURSE in a similar manner, leading to the hierarchy shown in part (3) . The COURSE1 replica to the left represents the relationship between courses and their sections, whereas the COURSE replica to the right represents the relationship between courses and the department that offers each course.

FIGURE 26.13
Converting a graph with cycles into a hierarchical
(tree) structure.



XML Querying

XPath

- An XPath expression returns a collection of element nodes that satisfy certain patterns specified in the expression.
- The names in the XPath expression are node names in the XML document tree that are either tag (element) names or attribute names, possibly with additional **qualifier conditions** to further restrict the nodes that satisfy the pattern.
- There are two main **separators** when specifying a path:
single slash (/) and double slash (//).
A single slash before a tag specifies that the tag must appear as a direct child of the previous (parent) tag, whereas a double slash specifies that the tag can appear as a descendant of the previous tag *at any level*.
- It is customary to include the file name in any XPath query allowing us to specify any local file name or path name that specifies the path.

doc(www.company.com/info.XML)/company => COMPANY XML doc

XML Querying

1. Returns the COMPANY root node and all its descendant nodes, which means that it returns the whole XML document.
2. Returns all department nodes (elements) and their descendant subtrees.
3. Returns all employeeName nodes that are direct children of an employee node, such that the employee node has another child element employeeSalary whose value is greater than 70000.
4. This returns the same result as the previous one except that we specified the full path name in this example.
5. This returns all projectWorker nodes and their descendant nodes that are children under a **path /company/project** and that have a child node hours with value greater than 20.0 hours.

FIGURE 26.14

Some examples of XPath expressions on XML documents
that follow the XML schema file COMPANY in Figure 26.5.

1. /company
2. /company/department
3. //employee [employeeSalary gt 70000]/employeeName
4. /company/employee [employeeSalary gt 70000]/employeeName
5. /company/project/projectWorker [hours ge 20.0]

XML Querying

XQuery

- XQuery uses XPath expressions, but has additional constructs.
- XQuery permits the specification of more general queries on one or more XML documents.
- The typical form of a query in XQuery is known as a **FLWR expression**, which stands for the four main clauses of XQuery and has the following form:

FOR <variable bindings to individual nodes (elements)>

LET <variable bindings to collections of nodes (elements)>

WHERE <qualifier conditions>

RETURN <query result specification>

XML Querying

1. This query retrieves the first and last names of employees who earn more than 70000. The variable \$x is bound to each employeeName element that is a child of an employee element, but only for employee elements that satisfy the qualifier that their **employeeSalary** is greater than 70000.
2. This is an alternative way of retrieving the same elements retrieved by the first query.
3. This query illustrates how a join operation can be performed by having more than one variable. Here, the \$x variable is bound to each projectWorker element that is a child of project number 5, whereas the \$y variable is bound to each employee element. The join condition matches ssn values in order to retrieve the employee names.

FIGURE 26.15

Some examples of XQuery queries on XML documents that follow the XML schema file COMPANY in Figure 26.5.

1. FOR \$x IN
doc(www.company.com/info.xml)
//employee [employeeSalary gt 70000]/employeeName
RETURN <res> \$x/firstName, \$x/lastName </res>
2. FOR \$x IN
doc(www.company.com/info.xml)/company/employee
WHERE \$x/employeeSalary gt 70000
RETURN <res> \$x/employeeName/firstName,
\$x/employeeName/lastName </res>
3. FOR \$x IN
doc(www.company.com/info.xml)/company
/project[projectNumber = 5]/projectWorker,
\$y IN
doc(www.company.com/info.xml)/company/employee
WHERE \$x/hours gt 20.0 AND \$y.ssn = \$x.ssn
RETURN <res> \$y/employeeName/firstName,
\$y/employeeName/lastName, \$x/hours </res>

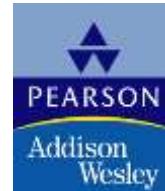
Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 27

Data Mining Concepts



Copyright © 2004 Pearson Education, Inc.

Overview of Data Mining Technology

Association Rules

- Market-Basket Model, Support, and Confidence
- Apriori Algorithm
- Sampling Algorithm
- Frequent-Pattern Tree Algorithm
- Partition Algorithm
- Other Types of Association Rules
- Additional Considerations for Association Rules

Classification

Clustering

Approaches to Other Data Mining Problems

- Discovery of Sequential Patterns
- Discovery of Patterns in Time Series
- Regression
- Neural Networks
- Genetic Algorithm

Applications of Data Mining

Commercial Data Mining Tools

Summary

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 28

Overview of Data Warehousing and OLAP



Copyright © 2004 Pearson Education, Inc.

Introduction, Definitions, and Terminology

Characteristics of Data Warehouses

Data Modeling for Data Warehouses

Building a Data Warehouse

Typical Functionality of a Data Warehouse

Data Warehouse Versus Views

Problems and Open Issues in Data Warehouses

- Difficulties of Implementing Data Warehouses
- Open Issues in Data Warehousing

Summary

Fundamentals of
**DATABASE
SYSTEMS**

FOURTH EDITION

ELMASRI  NAVATHE

Chapter 29

Emerging Database Technologies and Applications



Chapter Outline

1 Mobile Databases

- 1.1 Mobile Computing Architecture
- 1.2 Characteristics of Mobile Environments
- 1.3 Data Management Issues
- 1.4 Application: Intermittently Synchronized Databases

2 Multimedia Databases

- 2.1 The Nature of Multimedia Data and Applications
- 2.2 Data Management Issues
- 2.3 Open Research Problems
- 2.4 Multimedia Database Applications

Chapter Outline(contd.)

3 Geographic Information Systems

3.1 GIS Applications

3.2 Data Management Requirements of GIS

3.3 Specific GIS Data Operations

3.4 An Example of GIS Software: ARC-INFO

3.5 Problems and Future issues in GIS

Chapter Outline(contd.)

4 GENOME Data Management

4.1 Biological Sciences and Genetics

4.2 Characteristics of Biological Data

4.3 The Human Genome Project and Existing Biological Databases

Emerging Database Technologies and Applications

- Emerging database technologies
- The major application domains

1 Mobile Databases

Recent advances in portable and wireless technology led to **mobile computing**, a new dimension in data communication and processing.

Portable computing devices coupled with wireless communications allow clients to access data from virtually anywhere and at any time.

1 Mobile Databases(2)

There are a number of hardware and software problems that must be resolved before the capabilities of mobile computing can be fully utilized.

Some of the software problems – which may involve data management, transaction management, and database recovery – have their origins in distributed database systems.

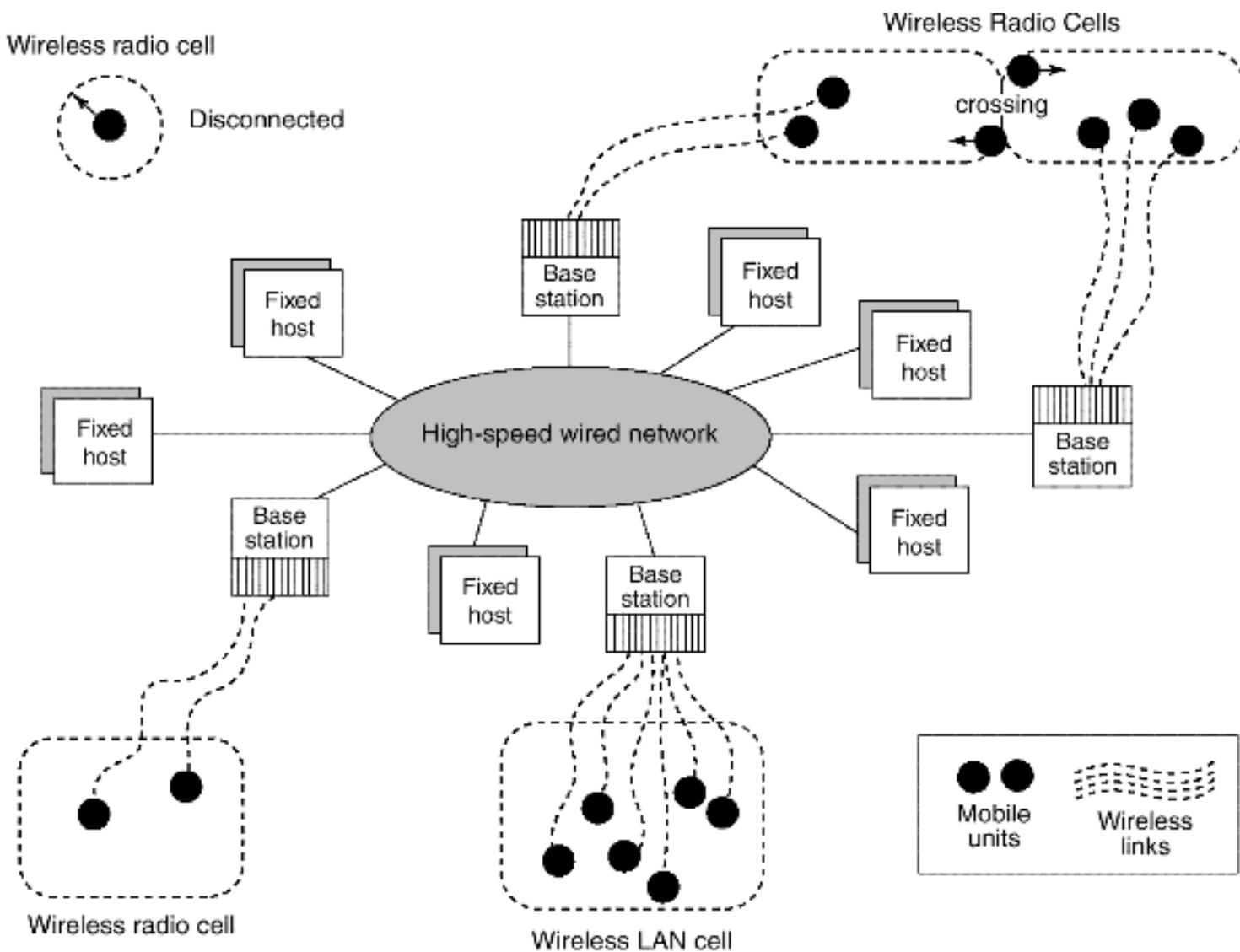
1 Mobile Databases(3)

In mobile computing, the problems are more difficult, mainly:

- The limited and intermittent connectivity afforded by wireless communications.
- The limited life of the power supply(battery).
- The changing topology of the network.
 - In addition, mobile computing introduces new architectural possibilities and challenges.

1.1 Mobile Computing Architecture

The general architecture of a mobile platform is illustrated in Fig29.1.



1.1 Mobile Computing Architecture(2)

It is distributed architecture where a number of computers, generally referred to as **Fixed Hosts** and **Base Stations** are interconnected through a high-speed wired network.

- Fixed hosts are general purpose computers configured to manage mobile units.
- Base stations function as gateways to the fixed network for the **Mobile Units**.

1.1 Mobile Computing Architecture(3)

Wireless Communications –

- The wireless medium have bandwidth significantly lower than those of a wired network.
 - The current generation of wireless technology has data rates range from the tens to hundreds of kilobits per second (2G cellular telephony) to tens of megabits per second (wireless Ethernet, popularly known as WiFi).
 - Modern (wired) Ethernet, by comparison, provides data rates on the order of hundreds of megabits per second.

1.1 Mobile Computing Architecture(4)

Wireless Communications –

- The other characteristics distinguish wireless connectivity options:
 - interference,
 - locality of access,
 - range,
 - support for packet switching,
 - seamless roaming throughout a geographical region.

1.1 Mobile Computing Architecture(5)

Wireless Communications –

- Some wireless networks, such as WiFi and Bluetooth, use unlicensed areas of the frequency spectrum, which may cause interference with other appliances, such as cordless telephones.
- Modern wireless networks can transfer data in units called packets, that are used in wired networks in order to conserve bandwidth.

1.1 Mobile Computing Architecture(6)

Client/Network Relationships –

- Mobile units can move freely in a **geographic mobility domain**, an area that is circumscribed by wireless network coverage.
 - To manage entire mobility domain is divided into one or more smaller domains, called **cells**, each of which is supported by at least one base station.
 - Mobile units be unrestricted throughout the cells of domain, while maintaining information **access contiguity**.

1.1 Mobile Computing Architecture(7)

Client/Network Relationships –

The communication architecture described earlier is designed to give the mobile unit the impression that it is attached to a fixed network, emulating a traditional client-server architecture.

Wireless communications, however, make other architectures possible. One alternative is a mobile ad-hoc network (MANET), illustrated in 29.2.

1.1 Mobile Computing Architecture(8)

1.1 Mobile Computing Architecture(9)

Client/Network Relationships –

- In a MANET, co-located mobile units do not need to communicate via a fixed network, but instead, form their own using cost-effective technologies such as Bluetooth.
- In a MANET, mobile units are responsible for routing their own data, effectively acting as base stations as well as clients.
 - Moreover, they must be robust enough to handle changes in the network topology, such as the arrival or departure of other mobile units.

1.1 Mobile Computing Architecture(10)

Client/Network Relationships –

- MANET applications can be considered as peer-to-peer, meaning that a mobile unit is simultaneously a client and a server.
 - Transaction processing and data consistency control become more difficult since there is no central control in this architecture.
 - Resource discovery and data routing by mobile units make computing in a MANET even more complicated.
 - Sample MANET applications are multi-user games, shared whiteboard, distributed calendars, and battle information sharing.

1.2 Characteristics of Mobile Environments

The characteristics of mobile computing include:

- Communication latency.
- Intermittent connectivity.
- Limited battery life.
- Changing client location.

1.2 Characteristics of Mobile Environments(2)

The server may not be able to reach a client. A client may be unreachable because it is **dozing** – in an energy-conserving state in which many subsystems are shut down – or because it is out of range of a base station.

In either case, neither client nor server can reach the other, and modifications must be made to the architecture in order to compensate for this case.

Proxies for unreachable components are added to the architecture. For a client (and symmetrically for a server), the proxy can cache updates intended for the server.

1.2 Characteristics of Mobile Environments(3)

Mobile computing poses challenges for servers as well as clients. The latency involved in wireless communication makes scalability a problem. Because latency due to wireless communications increases the time to service each client request, the server can handle fewer clients. One way servers relieve this problem is by **broadcasting** data whenever possible.

- A server can simply broadcast data periodically.
- Broadcast also reduces the load on the server, as clients do not have to maintain active connections to it.

1.2 Characteristics of Mobile Environments(4)

Client mobility also poses many data management challenges.

- Servers must keep track of client locations in order to efficiently route messages to them.
- Client data should be stored in the network location that minimizes the traffic necessary to access it.
- The act of moving between cells must be transparent to the client.
- The server must be able to gracefully divert the shipment of data from one base to another, without the client noticing.
- Client mobility also allows new applications that are *location-based*.

1.3 Data Management Issues

From a data management standpoint, mobile computing may be considered a variation of distributed computing.

Mobile databases can be distributed under two possible scenarios:

1. The entire database is distributed mainly among the wired components, possibly with full or partial replication. A base station or fixed host manages its own database with a DBMS-like functionality, with additional functionality for locating mobile units and additional query and transaction management features to meet the requirements of mobile environments.
2. The database is distributed among wired and wireless components. Data management responsibility is shared among base stations or fixed hosts and mobile units.

1.3 Data Management Issues(2)

Data management issues as it is applied to mobile databases:

- Data distribution and replication
- Transactions models
- Query processing
- Recovery and fault tolerance
- Mobile database design
- Location-based service
- Division of labor
- Security

1.4 Application: Intermittently Synchronized Databases

Whenever clients connect – through a process known in industry as *synchronization* of a client with a server – they receive a batch of updates to be installed on their local database. The primary characteristic of this scenario is that the clients are mostly disconnected; the server is not necessarily able reach them. This environment has problems similar to those in distributed and client-server databases, and some from mobile databases.

This environment is referred to as **Intermittently Synchronized Database Environment (ISDBE)**.

1.4 Application: Intermittently Synchronized Databases(2)

The characteristics of **Intermittently Synchronized Databases** (ISDBs) make them distinct from the mobile databases are:

1. A client connects to the server when it wants to exchange updates. The communication can be *unicast* – one-on-one communication between the server and the client– or *multicast*– one sender or server may periodically communicate to a set of receivers or update a group of clients.
2. A server cannot connect to a client at will.

1.4 Application: Intermittently Synchronized Databases(3)

3. Issues of wireless versus wired client connections and power conservation are generally immaterial.
4. A client is free to manage its own data and transactions while it is disconnected. It can also perform its own recovery to some extent.
5. A client has multiple ways connecting to a server and, in case of many servers, may choose a particular server to connect to based on proximity, communication nodes available, resources available, etc.

2 Multimedia Databases

In the years ahead multimedia information systems are expected to dominate our daily lives. Our houses will be wired for bandwidth to handle interactive multimedia applications. Our high-definition TV/computer workstations will have access to a large number of databases, including digital libraries, image and video databases that will distribute vast amounts of multisource multimedia content.

2.1 Multimedia Databases

DBMSs have been constantly adding to the types of data they support. Today the following types of multimedia data are available in current systems.

- *Text:* May be formatted or unformatted. For ease of parsing structured documents, standards like SGML and variations such as HTML are being used.
- *Graphics:* Examples include drawings and illustrations that are encoded using some descriptive standards (e.g. CGM, PICT, postscript).

2.1 Multimedia Databases(2)

- *Images*: Includes drawings, photographs, and so forth, encoded in standard formats such as bitmap, JPEG, and MPEG. Compression is built into JPEG and MPEG. These images are not subdivided into components. Hence querying them by content (e.g., find all images containing circles) is nontrivial.
- *Animations*: Temporal sequences of image or graphic data.

2.1 Multimedia Databases(3)

- *Video*: A set of temporally sequenced photographic data for presentation at specified rates— for example, 30 frames per second.
- Structured *audio*: A sequence of audio components comprising note, tone, duration, and so forth.
- *Audio*: Sample data generated from aural recordings in a string of bits in digitized form. Analog recordings are typically converted into digital form before storage.

2.1 Multimedia Databases(4)

- *Composite or mixed multimedia data:* A combination of multimedia data types such as audio and video which may be physically mixed to yield a new storage format or logically mixed while retaining original types and formats. Composite data also contains additional control information describing how the information should be rendered.

2.1 Multimedia Databases(5)

Nature of Multimedia Applications: Multimedia data may be stored, delivered, and utilized in many different ways. Applications may be categorized based on their data management characteristics as follows:

- *Repository applications*: A large amount of multimedia data as well as metadata is stored for retrieval purposes. Examples include repositories of satellite images, engineering drawings and designs, space photographs, and radiology scanned pictures.

2.1 Multimedia Databases(6)

- *Presentation applications:* A large amount of applications involve delivery of multimedia data subject to temporal constraints; simple multimedia viewing of video data, for example, requires a system to simulate VCR-like functionality. Complex and interactive multimedia presentations involve orchestration directions to control the retrieval order of components in a series or in parallel. Interactive environments must support capabilities such as real-time editing analysis or annotating of video and audio data.

2.1 Multimedia Databases(7)

- *Collaborative work using multimedia information:* This is a new category of applications in which engineers may execute a complex design task by merging drawings, fitting subjects to design constraints, and generating new documentation, change notifications, and so forth. Intelligent healthcare networks as well as telemedicine will involve doctors collaborating among themselves, analyzing multimedia patient data and information in real time as it is generated.

2.2 Data Management Issues

Multimedia applications dealing with thousands of images, documents, audio and video segments, and free text data depend critically on appropriate modeling of the structure and content of data and then designing appropriate database schemas for storing and retrieving multimedia information. Multimedia information systems are very complex and embrace a large set of issues :

- *Modeling*
 - complex objects

2.2 Data Management Issues(2)

- *Design*
 - conceptual, logical, and physical design of multimedia has not been addressed fully.
- *Storage*
 - multimedia data on standard disklike devices presents problems of representation, compression, mapping to device hierarchies, archiving, and buffering during the input/output operation.
- *Queries and retrieval*
 - “database” way of retrieving information is based on query languages and internal index structures.

2.2 Data Management Issues(3)

- *Performance*
 - multimedia applications involving only documents and text, performance constraints are subjectively determined by the user.
 - applications involving video playback or audio-video synchronization, physical limitations dominate.

2.3 Multimedia Database Applications

Large-scale applications of multimedia databases can be expected to encompass a large number of disciplines and enhance existing capabilities.

- Documents and records management
- Knowledge dissemination
- Education and training
- Marketing, advertising, retailing, entertainment, and travel
- Real-time control and monitoring

3 Geographic Information Systems

Geographic information systems(GIS) are used to collect, model, and analyze information describing physical properties of the geographical world. The scope of GIS broadly encompasses two types of data:

1. spatial data, originating from maps, digital images, administrative and political boundaries, roads, transportation networks, physical data, such as rivers, soil characteristics, climatic regions, land elevations, and

3 Geographic Information Systems(2)

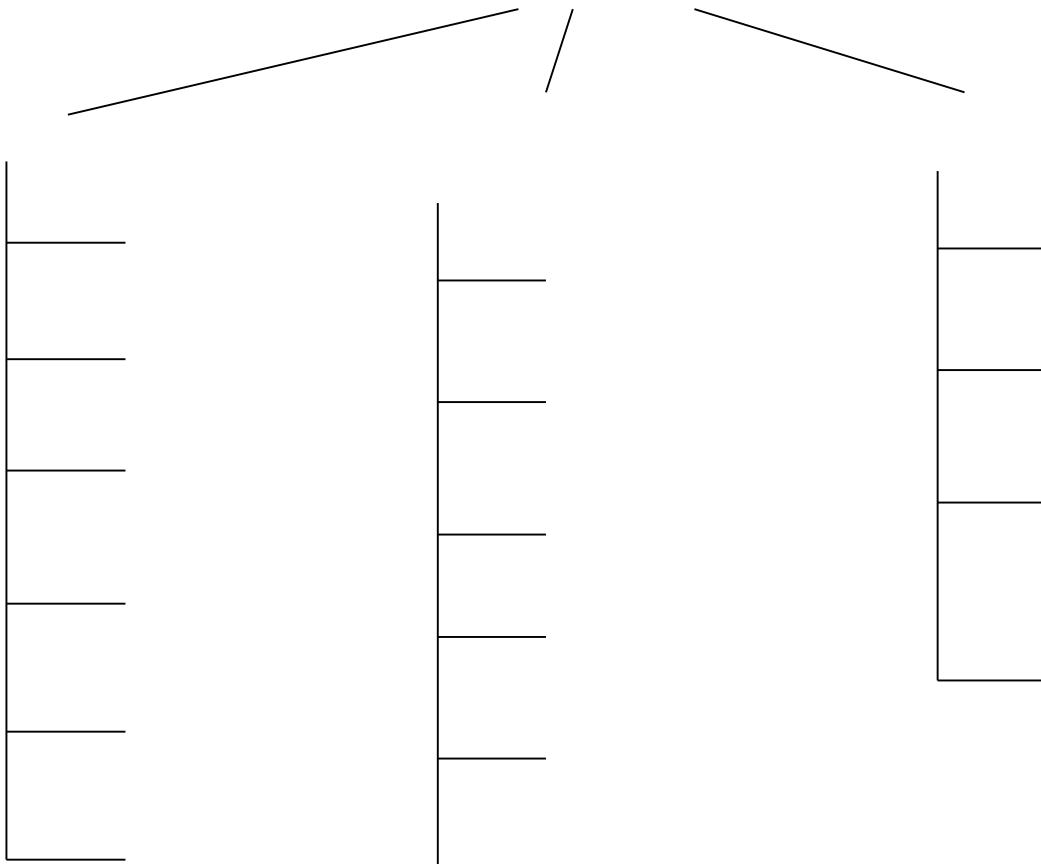
2. nonspatial data, such as socio-economic data (like census counts), economic data, and sales or marketing information. GIS is a rapidly developing domain that offers highly innovative approaches to meet some challenging technical demands.

3.1 GIS Applications

It is possible to divide GISs into three categories:

- cartographic applications,
- digital terrain modeling applications, and
- geographic objects applications

3.1 GIS Applications(2)



3.2 Data Management Requirements of GIS

The functional requirements of the GIS applications above translate into the following database requirements.

Data Modeling and Representation, GIS data can be broadly represented in two formats:

1. **Vector** data represents geometric objects such as points, lines, and polygons.

3.2 Data Management Requirements of GIS(2)

2. **Raster** data is characterized as an array of points, where each point represents the value of an attribute for a real-world location. Informally, raster images are n-dimensional array where each entry is a unit of the image and represents an attribute. Two-dimensional units are called *pixels*, while three-dimensional units are called *voxels*. Three-dimensional elevation data is stored in a raster-based **digital elevation model (DEM)** format.

3.2 Data Management Requirements of GIS(3)

Another raster format called triangular irregular network (TIN) is a topological vector-based approach that models surfaces by connecting sample points as vector of triangles and has a point density that may vary with the roughness of the terrain. rectangular grids (or elevation matrices) are two-dimensional array structures. In **digital terrain modeling** (DTM), the model also may be used by substituting the elevation with some attribute of interest such as population density or air temperature. GIS data often includes a temporal structure in addition to a spatial structure.

3.2 Data Management Requirements of GIS(4)

Data Analysis, GIS data undergoes various types of analysis.

For example, in applications such as soil erosion studies, environmental impact studies, or hydrological runoff simulations, DTM data may undergo various types of **geomorphometric analysis** – measurements such as slope values, *gradients* (the rate of change in altitude), *aspect* (the compass direction of the gradient), *profile convexity* (the rate of change of gradient), *plan convexity* (the convexity of contours and other parameters).

3.2 Data Management Requirements of GIS(5)

Data Integration, GISs must integrate both vector and raster data from a variety of sources. Sometimes edges and regions are inferred from a raster image to form a vector model, or conversely, raster images such as aerial photographs are used to update vector models. Several coordinate systems such as Universal Transverse Mercator (UTM), latitude/longitude, and local cadastral systems are used to identify locations. Data originating from different coordinate systems requires appropriate transformations.

3.2 Data Management Requirements of GIS(6)

Data Capture, The first step in developing a spatial database for cartographic modeling is to capture the two-dimensional or three-dimensional geographical information in digital form – a process that is sometimes impeded by source map characteristics such as resolution, type of projection, map scales, cartographic licensing, diversity of measurement techniques, and coordinate system differences. Spatial data can also be captured from remote sensors in satellites such as Landsat, NORA, and Advanced Very High Resolution Radiometer(AVHRR) as well as SPOT HRV (High Resolution Visible Range Instrument).

3.3 Specific GIS Data Operations

GIS applications are conducted through the use of special operators such as the following:

1. *Interpolation*
2. *Interpretation*
3. *Proximity analysis*
4. *Raster image processing*
5. *Analysis of networks*

3.3 Specific GIS Data Operations(2)

The functionality of a GIS database is also subject to other considerations:

1. *Extensibility*
2. *Data quality control*
3. *Visualization*

Such requirements clearly illustrate that standard RDBMSs or ODBMSs do not meet the special needs of GIS. It is therefore necessary to design systems that support the vector and raster representations and the spatial functionality as well as the required DBMS features.

4.1 Genome Data Management

Biological Sciences and Genetics: The biological sciences encompass an enormous variety of information.

Environmental science gives us a view of how species live and interact in a world filled with natural phenomena. Biology and ecology study particular species. Anatomy focuses on the overall structure of an organism, documenting the physical aspects of individual bodies. Traditional medicine and physiology break the organism into systems and tissues and strive to collect information on the workings of these systems and the organism as a whole.

4.1 Genome Data Management(2)

Histology and cell biology delve into the tissue and cellular levels and provide knowledge about the inner structure and function of the cell. This wealth of information that has been generated, classified, and stored for centuries has only recently become a major application of database technology.

4.1 Genome Data Management(3)

Genetics has emerged as an ideal field for the application of information technology. In a broad sense, it can be taught of as the construction of models based on information about genes – which can be defined as units of heredity – and population and the seeking out of relationships in that information.

4.1 Genome Data Management(4)

The study of genetics can be divided into three branches:

1. Mendelian genetics is the study of the transmission of traits between generations.
2. Molecular genetics is the study of the chemical structure and function of genes at the molecular level.
3. Population genetics is the study of how genetic information varies across populations of organisms.

4.1 Genome Data Management(5)

The origins of molecular genetics can be traced to two important discoveries:

1. In 1869 when Friedrich Miescher discovered nuclein and its primary component, deoxyribonucleic acid (DNA). In subsequent research DNA and a related compound, ribonucleic acid , were found to be composed of nucleotides (a sugar, a phosphate, and a base which combined to form nucleic acid) linked into long polymers via the sugar and phosphate.

4.1 Genome Data Management(6)

2. The second discovery was the demonstration in 1944 by Oswald Avery that DNA was indeed the molecular substance carrying genetic information.

4.1 Genome Data Management(7)

Genes were shown to be composed of chains of nucleic acids arranged linearly on chromosomes and to serve three primary functions:

1. replicating genetic information between generations,
2. providing blueprints for the creation of polypeptides, and
3. accumulating changes—thereby allowing evolution to occur.

Watson and Crick found the double-helix structure of the DNA in 1953, which gave molecular biology a new direction.

4.2 Characteristics of Biological Data

Biological data exhibits many special characteristics that make management of biological information a particularly challenging problem. The characteristics related to biological information, and focusing on a multidisciplinary field called **bioinformatics** that has emerged. Bioinformatics addresses information management of genetic information with special emphasis on DNA sequence analysis.

4.2 Characteristics of Biological Data(2)

Applications of bioinformatics span design of targets for drugs, study of mutations and related diseases, anthropological investigations on migration patterns of tribes and therapeutic treatments.

Characteristic 1: Biological data is highly complex when compared with most other domains or applications.

Characteristic 2: The amount and range of variability in data is high.

Characteristic 3: Schemas in biological databases change at a rapid pace.

4.2 Characteristics of Biological Data(3)

Characteristic 4: Representations of the same data by different biologists will likely be different (even using the same system).

Characteristic 5: Most users of biological data do not require write access to the database; read-only access is adequate.

Characteristic 6: Most biologists are not likely to have knowledge of the internal structure of the database or about schema design.

4.2 Characteristics of Biological Data(4)

Characteristic 7: The context of data gives added meaning for its use in biological applications.

Characteristic 8: Defining and representing complex queries is extremely important to the biologist.

Characteristic 9: Users of biological information often require access to “old” values of the data – particularly when verifying previously reported results.

4.3 The Human Genome Project and Existing Biological Databases

The term *genome* is defined as the total genetic information that can be obtained about an entity. The **human genome**, for example, generally refers to the complete set of genes required to create a human being – estimated to be more than 30,000 genes spread over 23 pairs of chromosomes, with an estimated 3 to 4 billion nucleotides. The goal of the Human Genome Project(HGP) has been to obtain the complete sequence – the ordering of the bases – of those nucleotides.

4.3 The Human Genome Project and Existing Biological Databases

The term *genome* is defined as the total genetic information that can be obtained about an entity. The **human genome**, for example, generally refers to the complete set of genes required to create a human being – estimated to be more than 30,000 genes spread over 23 pairs of chromosomes, with an estimated 3 to 4 billion nucleotides. The goal of the Human Genome Project(HGP) has been to obtain the complete sequence – the ordering of the bases – of those nucleotides.

4.3 The Human Genome Project and Existing Biological Databases(2)

Some of the existing database systems that are supporting or have grown out of the Human Genome Project.

GenBank – The preeminent DNA sequence database in the world today is GenBank, maintained by the National Center for Biotechnology Information (NCBI) of the National Library of Medicine (NLM).

4.3 The Human Genome Project and Existing Biological Databases(3)

GenBank –

- Established in 1978 as a repository for DNA sequence data.
- Since 1978 expanded to include sequence tag data, protein sequence data, three-dimensional protein structure, taxonomy, and links to the medical literature (MEDLINE).

4.3 The Human Genome Project and Existing Biological Databases(4)

GenBank –

- As of release 135.0 in April 2003, GenBank contains over 31 billion nucleotide bases of more than 24 million sequences from over 100,000 species with roughly 1400 new organisms being added each month.
- The database size in flat file format is over 100 GB uncompressed and has been doubling every 15 months.
- International collaboration with the European Molecular Biology Laboratory (EMBL) in the U.K. and the DNA Data Bank of Japan (DDBJ) on daily basis.

4.3 The Human Genome Project and Existing Biological Databases(5)

GenBank –

- Other limited data sources (e.g. three-dimensional structure and Online Mendelian Inheritance in Man (OMIM), have been added recently by reformatting the existing OMIM and PDB databases and redesigning the structure of the GenBank system to accommodate these new data sets.
- The system is maintained as a combination of flat files, relational databases, and files containing Abstract Syntax Notation One (ASN.1)

4.3 The Human Genome Project and Existing Biological Databases(6)

GenBank –

- The average user of the database is not able to access the structure of the data directly for querying or other functions, although complete snapshots of the database are available for export in a number of formats, including ASN.1. The query mechanism provided is via the Entrez application (or its www version), which allows keyword, sequence, and GenBank UID searching through a static interface.

4.3 The Human Genome Project and Existing Biological Databases(7)

The Genome Database (GDB) –

- Created in 1989, GDB is a catalog of human gene mapping data, a process that associates a piece of information with a particular location on the human genome.
- GDB data includes data describing primarily map information (distance and confidence limits), and Polymerase Chain Reaction (PCR) probe data (experimental conditions, PCR primers, and reagents used).

4.3 The Human Genome Project and Existing Biological Databases(8)

The Genome Database (GDB) –

- More recently efforts have been made to add data on mutations linked to genetic loci, cell lines used in experiments, DNA probe libraries, and some limited polymorphism and population data.
- The GDB system is built around SYBASE, a commercial relational DBMS, and its data are modeled using standard Entity-Relationship techniques.
 - GDB distributes a Database Access Toolkit

4.3 The Human Genome Project and Existing Biological Databases(9)

The Genome Database (GDB) –

- As with GenBank, users are given only a very high-level view of the data at the time of searching and thus cannot make use of any knowledge gleaned from the structure of the GDB tables. Search methods are most useful when users are simply looking for an index into map or probe data. Exploratory ad hoc searching is not encouraged by present interfaces. Integration of the database structures of GDB and OMIM was never fully established.

4.3 The Human Genome Project and Existing Biological Databases(10)

Online Mendelian Inheritance in Man –

- Online Mendelian Inheritance in Man (OMIM) is an electronic compendium of information on the genetic basis of human disease.
- Begun in hard-copy form by Victor McCusick in 1966 with 1500 entries, it was converted to a full-text electronic form between 1987 and 1989 by GDB.
 - In 1991 its administration was transferred from John Hopkins University to the NCBI, and the entire database was converted to NCBI's GenBank format. Today it contains more than 14,000 entries.

4.3 The Human Genome Project and Existing Biological Databases(11)

Online Mendelian Inheritance in Man –

- OMIM covers material on five disease areas based loosely on organs and systems. Any morphological, biochemical, behavioral, or other properties under study are referred to as **phenotype** of an individual (or a cell). Mendel realized that genes can exist in numerous forms known as **alleles**. A **genotype** refers to the actual allelic composition of an individual.

4.3 The Human Genome Project and Existing Biological Databases(12)

EcoCyc. –

- The Encyclopedia of *Escherichia coli* Genes and Metabolism (EcoCyc) is a recent experiment in combining information about the genome and the metabolism of *E.coli* K-12.
- The database was created in 1996 as a collaboration between Stanford Research Institute and Marine Biological Laboratory.

4.3 The Human Genome Project and Existing Biological Databases(13)

EcoCyc. –

- An object-oriented data model was first used to implement the system, with data stored in Ocelot, a frame knowledge representation system. EcoCyc data was arranged in a hierarchy of object classes based on observations that
 - the properties of a reaction are independent of an enzyme that catalyzes it, and
 - an enzyme has a number of properties that are “logically distinct” from its reactions.

4.3 The Human Genome Project and Existing Biological Databases(14)

EcoCyc. –

- EcoCyc provides two methods of querying:
 - direct (via predefined queries) and
 - indirect (via hypertext navigation).

4.3 The Human Genome Project and Existing Biological Databases(15)

Gene Ontology –

- Gene Ontology (GO) Consortium was formed in 1998 as a collaboration among three model organism databases: FlyBase, Mouse Genome Informatics (MGI) and Saccharomyces or yeast Genome Database (SGD).
 - goal is to produce a structured, precisely defined, common, controlled vocabulary for describing the roles of genes and gene products in any organism.

4.3 The Human Genome Project and Existing Biological Databases(16)

Gene Ontology –

- With the completion of genome sequencing of many species, it has been observed that a large fraction of genes among organisms display similarity in biological roles and biologists have acknowledged that there is likely to be a single limited universe of genes and proteins that are conserved in most or all living cells.
- The GO Consortium has developed three ontologies: Molecular function, biological process, and cellular component, to describe attributes of genes, gene products, or gene product groups.

4.3 The Human Genome Project and Existing Biological Databases(17)

Gene Ontology –

- Each ontology comprises a set of well-defined vocabularies of terms and relationships.
 - The terms are organized in the form of directed acyclic graphs (DAGs), in which a term node may have multiple parents and multiple children.
 - A child term can be an instance of (*is a*) or *part of* its parent.
 - Latest release of GO database has over 13,000 terms and more than 18,000 relationships between terms.
 - GO was implemented using MySQL, an open source relational database and a monthly database release is available in SQL and XML formats.

Summary Of the Major Genome-Related Databases

DATABASE NAME	MAJOR CONTENT	INITIAL TECHNOLOGY	CURRENT TECHNOLOGY	DB PROBLEM AREAS	PRIMARY DATA TYPES
GenBank	DNA/RNA sequence, protein	Text files	Flat-file/ASN.1	Schema browsing, schema evolution, linking to other dbs	Text, numeric, Some complex types
OMIM	Disease phenotypes and genotypes,etc	Index cards/text files	Flat-file/ASN.1	Unstructured, free text entries linking to other dbs	Text
GDB	Genetic map linkage data	Flat file	Relational	Schema expansion / evolution, complex objects, linking to other dbs	Text, Numeric
ACEDB	Genetic map linkage data, sequence data(non-human)	OO	OO	Schema expansion /evolution, linking to other dbs	Text, Numeric
HGMDB	Sequence and sequence variants	Flat File-application specific	Flat File-application specific	Schema expansion /evolution, linking to other dbs	Text
EcoCyc	Biochemical reactions and pathways	OO	OO	Locked into class hierarchy, schema evolution	Complex types, text, numeric