

CHAPTER 1

Databases and Database Users

OUTLINE

- Types of Databases and Database Applications
- Basic Definitions
- Typical DBMS Functionality
- Example of a Database (UNIVERSITY)
- Main Characteristics of the Database Approach
- Types of Database Users
- Advantages of Using the Database Approach
- Historical Development of Database Technology
- Extending Database Capabilities
- When Not to Use Databases

Types of Databases and Database Applications

- Traditional Applications:
 - Numeric and Textual Databases
- More Recent Applications:
 - Multimedia Databases
 - Geographic Information Systems (GIS)
 - Biological and Genome Databases
 - Data Warehouses
 - Mobile databases
 - Real-time and Active Databases
- First part of book focuses on traditional applications
- *A number of recent applications are described later in the book (for example, Chapters 24,25,26,27,28,29)*

Recent Developments (1)

- Social Networks started capturing a lot of information about people and about communications among people-posts, tweets, photos, videos in systems such as:
 - Facebook
 - Twitter
 - Linked-In
- All of the above constitutes data
- Search Engines- Google, Bing, Yahoo : collect their own repository of web pages for searching purposes

Recent Developments (2)

- New Technologies are emerging from the so-called non-database software vendors to manage vast amounts of data generated on the web:
- Big Data storage systems involving large clusters of distributed computers (Chapter 25)
- NOSQL (Not Only SQL) systems (Chapter 24)
- A large amount of data now resides on the “cloud” which means it is in huge data centers using thousands of machines.

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.

Impact of Databases and Database Technology

- Businesses: Banking, Insurance, Retail, Transportation, Healthcare, Manufacturing
- Service Industries: Financial, Real-estate, Legal, Electronic Commerce, Small businesses
- Education : Resources for content and Delivery
- More recently: Social Networks, Environmental and Scientific Applications, Medicine and Genetics
- Personalized Applications: based on smart mobile devices

Simplified database system environment

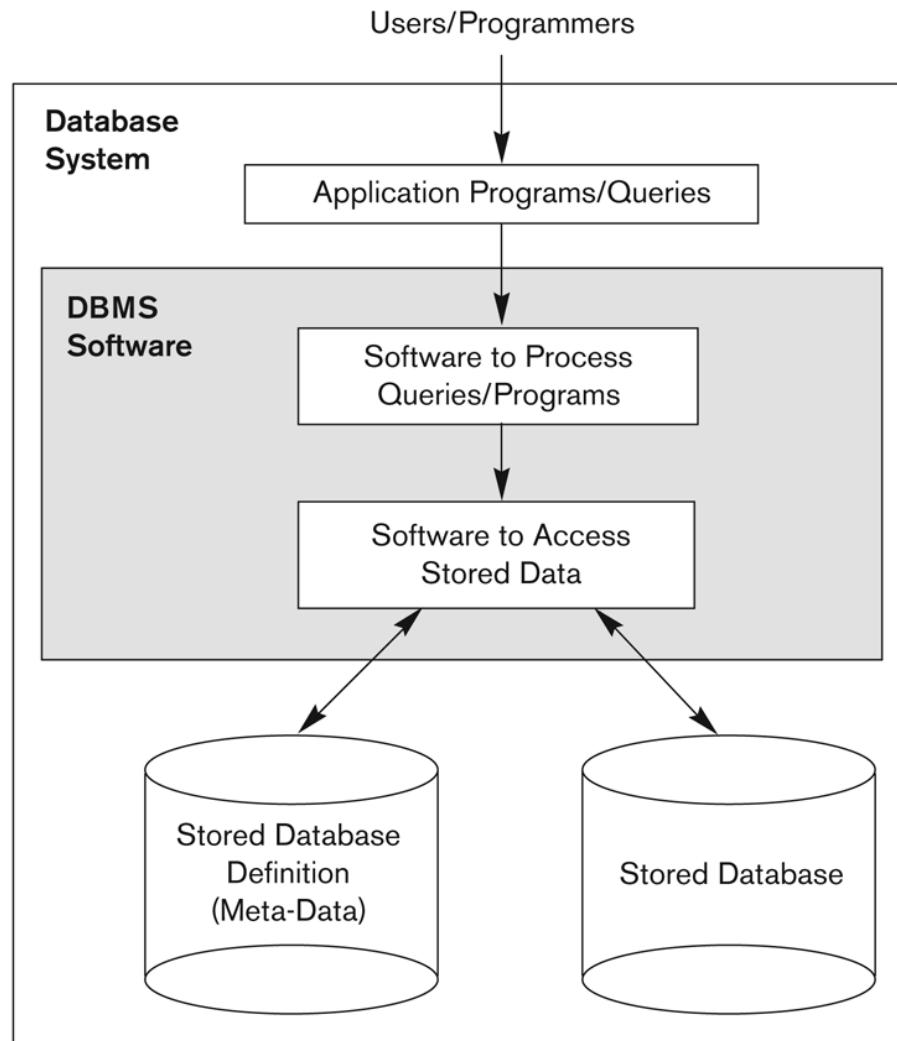


Figure 1.1
A simplified database system environment.

Typical DBMS Functionality

- *Define* a particular database in terms of its data types, structures, and constraints
- *Construct* or Load the initial database contents on a secondary storage medium
- *Manipulating* the database:
 - Retrieval: Querying, generating reports
 - Modification: Insertions, deletions and updates to its content
 - Accessing the database through Web applications
- *Processing* and *Sharing* by a set of concurrent users and application programs – yet, keeping all data valid and consistent

Application Activities Against a Database

- Applications interact with a database by generating
 - Queries: that access different parts of data and formulate the result of a request
 - Transactions: that may read some data and “update” certain values or generate new data and store that in the database
- Applications must not allow unauthorized users to access data
- Applications must keep up with changing user requirements against the database

Additional DBMS Functionality

- DBMS may additionally provide:
 - Protection or Security measures to prevent unauthorized access
 - “Active” processing to take internal actions on data
 - Presentation and Visualization of data
 - Maintenance of the database and associated programs over the lifetime of the database application
 - Called database, software, and system maintenance

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

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 entities and relationships are typically expressed in a conceptual data model, such as the ENTITY-RELATIONSHIP data model (see Chapters 3, 4)

Example of a simple database

COURSE

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

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

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

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2

A database that stores student and course information.

Main Characteristics of the Database Approach

- **Self-describing nature of a database system:**
 - A DBMS **catalog** stores the description of a particular database (e.g. data structures, types, and constraints)
 - The description is called **meta-data***.
 - This allows the DBMS software to work with different database applications.
- **Insulation between programs and data:**
 - Called **program-data independence**.
 - Allows changing data structures and storage organization without having to change the DBMS access programs.

* Some newer systems such as a few NOSQL systems need no meta-data: they store the data definition within its structure making it self describing

Example of a simplified database catalog

RELATIONS

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

COLUMNS

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

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

Figure 1.3

An example of a database catalog for the database in Figure 1.2.

Main Characteristics of the Database Approach (continued)

- **Data Abstraction:**
 - A **data model** is used to hide storage details and present the users with a conceptual view of the database.
 - Programs refer to the data model constructs rather than data storage details
- **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 (continued)

- **Sharing of data and multi-user transaction processing:**
 - Allowing a set of **concurrent users** to retrieve from and to update the database.
 - *Concurrency control* within the DBMS guarantees that each **transaction** is correctly executed or aborted
 - *Recovery* subsystem ensures each completed transaction has its effect permanently recorded in the database
 - **OLTP** (Online Transaction Processing) is a major part of database applications. This allows hundreds of concurrent transactions to execute per second.

Database Users

- Users may be divided into
 - Those who actually use and control the database content, and those who design, develop and maintain database applications (called “Actors on the Scene”), and
 - Those who design and develop the DBMS software and related tools, and the computer systems operators (called “Workers Behind the Scene”).

Database Users – Actors on the Scene

- Actors on the scene
 - **Database administrators:**
 - Responsible for authorizing access to the database, for coordinating 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.

Database End Users

- Actors on the scene (continued)
 - **End-users:** They use the data for queries, reports and some of them update the database content. End-users can be categorized into:
 - **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.
 - Users of Mobile Apps mostly fall in this category
 - Bank-tellers or reservation clerks are parametric users who do this activity for an entire shift of operations.
 - Social Media Users post and read information from websites

Database End Users (continued)

- **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 the user of a tax program that creates its own internal database.
- Another example is a user that maintains a database of personal photos and videos.

Database Users – Actors on the Scene (continued)

■ **System Analysts and Application Developers**

This category currently accounts for a very large proportion of the IT work force.

- **System Analysts:** They understand the user requirements of naïve and sophisticated users and design applications including canned transactions to meet those requirements.
- **Application Programmers:** Implement the specifications developed by analysts and test and debug them before deployment.
- **Business Analysts:** There is an increasing need for such people who can analyze vast amounts of business data and real-time data (“Big Data”) for better decision making related to planning, advertising, marketing etc.

Database Users – Actors behind the Scene

- **System Designers and Implementors:** Design and implement DBMS packages in the form of modules and interfaces and test and debug them. The DBMS must interface with applications, language compilers, operating system components, etc.
- **Tool Developers:** Design and implement software systems called tools for modeling and designing databases, performance monitoring, prototyping, test data generation, user interface creation, simulation etc. that facilitate building of applications and allow using database effectively.
- **Operators and Maintenance Personnel:** They manage the actual running and maintenance of the database system hardware and software environment.

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. Only the DBA staff uses privileged commands and facilities.
- Providing persistent storage for program Objects
 - E.g., Object-oriented DBMSs make program objects persistent— see Chapter 12.
- Providing Storage Structures (e.g. indexes) for efficient Query Processing – see Chapter 17.

Advantages of Using the Database Approach (continued)

- Providing optimization of queries for efficient processing.
- 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 from the stored data using deductive and active rules and triggers.

Additional Implications of Using the Database Approach

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

Additional Implications of Using the Database Approach (continued)

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

Historical Development of Database Technology

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

Historical Development of Database Technology (continued)

- Object-oriented and emerging applications:
 - Object-Oriented Database Management Systems (OODBMSs) were introduced in late 1980s and early 1990s to cater to the need of complex data processing in CAD and other applications.
 - Their use has not taken off much.
 - Many relational DBMSs have incorporated object database concepts, leading to a new category called *object-relational* DBMSs (ORDBMSs)
 - *Extended relational* systems add further capabilities (e.g. for multimedia data, text, XML, and other data types)

Historical Development of Database Technology (continued)

- 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). (see Ch. 13).
 - Script programming languages such as PHP and JavaScript allow generation of dynamic Web pages that are partially generated from a database (see Ch. 11).
 - Also allow database updates through Web pages

Extending Database Capabilities (1)

- New functionality is being added to DBMSs in the following areas:
 - Scientific Applications – Physics, Chemistry, Biology - Genetics
 - Earth and Atmospheric Sciences and Astronomy
 - XML (eXtensible Markup Language)
 - Image Storage and Management
 - Audio and Video Data Management
 - Data Warehousing and Data Mining – a very major area for future development using new technologies (see Chapters 28-29)
 - Spatial Data Management and Location Based Services
 - 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.

Extending Database Capabilities (2)

- Background since the advent of the 21st Century:
 - First decade of the 21st century has seen tremendous growth in user generated data and automatically collected data from applications and search engines.
 - Social Media platforms such as Facebook and Twitter are generating millions of transactions a day and businesses are interested to tap into this data to “understand” the users
 - Cloud Storage and Backup is making unlimited amount of storage available to users and applications

Extending Database Capabilities (3)

- Emergence of Big Data Technologies and NOSQL databases
 - New data storage, management and analysis technology was necessary to deal with the onslaught of data in petabytes a day (10^{15} bytes or 1000 terabytes) in some applications – this started being commonly called as “Big Data”.
 - Hadoop (which originated from Yahoo) and Mapreduce Programming approach to distributed data processing (which originated from Google) as well as the Google file system have given rise to Big Data technologies (Chapter 25). Further enhancements are taking place in the form of Spark based technology.
 - NOSQL (Not Only SQL- where SQL is the de facto standard language for relational DBMSs) systems have been designed for rapid search and retrieval from documents, processing of huge graphs occurring on social networks, and other forms of unstructured data with flexible models of transaction processing (Chapter 24).

When not to use a DBMS

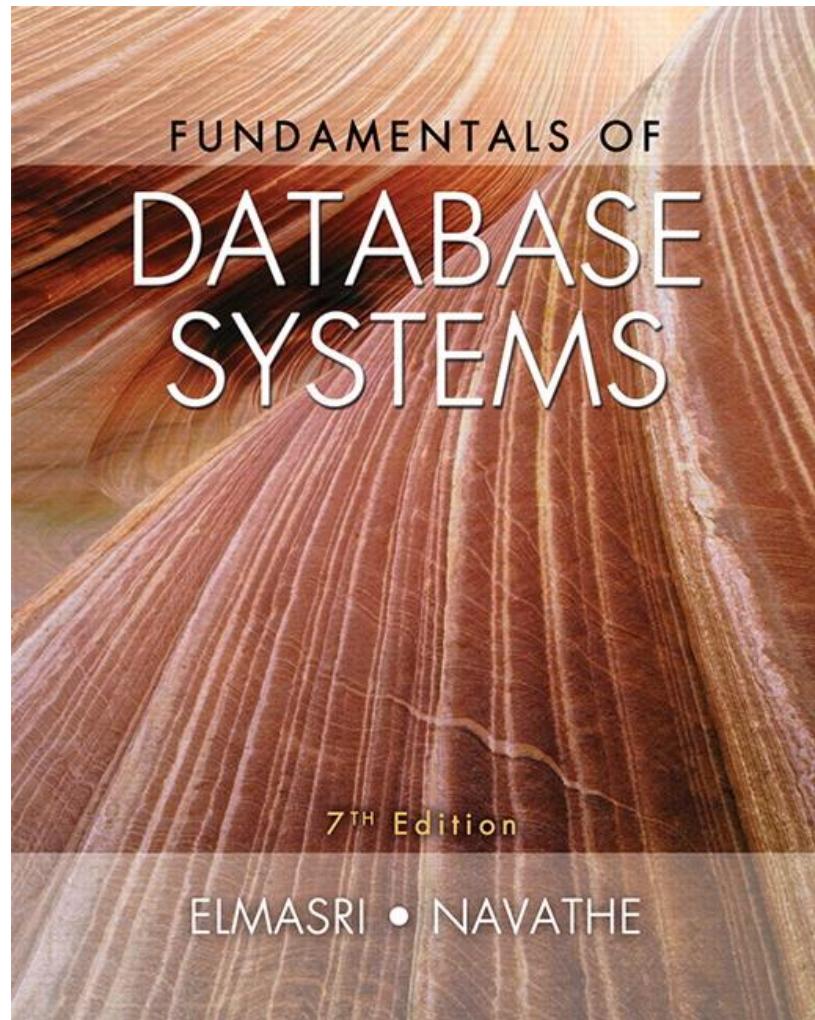
- Main inhibitors (costs) of using a DBMS:
 - High initial investment and possible need for additional hardware.
 - Overhead for providing generality, security, concurrency control, recovery, and integrity functions.
- When a DBMS may be unnecessary:
 - If the database and applications are simple, well defined, and not expected to change.
 - If access to data by multiple users is not required.
- When a DBMS may be infeasible:
 - In embedded systems where a general purpose DBMS may not fit in available storage

When not to use a DBMS

- When no DBMS may suffice:
 - If there are stringent real-time requirements that may not be met because of DBMS overhead (e.g., telephone switching systems)
 - If the database system is not able to handle the complexity of data because of modeling limitations (e.g., in complex genome and protein databases)
 - If the database users need special operations not supported by the DBMS (e.g., GIS and location based services).

Chapter Summary

- Types of Databases and Database Applications
- Basic Definitions
- Typical DBMS Functionality
- Example of a Database (UNIVERSITY)
- Main Characteristics of the Database Approach
- Types of Database Users
- Advantages of Using the Database Approach
- Historical Development of Database Technology
- Extending Database Capabilities
- When Not to Use Databases



CHAPTER 2

Database System Concepts and Architecture

Outline

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

Data Models

- **Data Model:**
 - A set of concepts to describe the ***structure*** of a database, the ***operations*** for manipulating these structures, and certain ***constraints*** that the database should obey.
- **Data Model Structure and Constraints:**
 - Constructs are used to define the database structure
 - Constructs typically include ***elements*** (and their ***data types***) as well as groups of elements (e.g. ***entity, record, table***), and ***relationships*** among such groups
 - Constraints specify some restrictions on valid data; these constraints must be enforced at all times

Data Models (continued)

■ Data Model Operations:

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

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. These are usually specified in an ad-hoc manner through DBMS design and administration manuals
- **Implementation (representational) data models:**
 - Provide concepts that fall between the above two, used by many commercial DBMS implementations (e.g. relational data models used in many commercial systems).
- **Self-Describing Data Models:**
 - Combine the description of data with the data values. Examples include XML, key-value stores and some NOSQL systems.

Schemas versus Instances

- Database Schema:
 - The ***description*** of a database.
 - Includes descriptions of the database structure, data types, and the constraints on the database.
- Schema Diagram:
 - An ***illustrative*** display of (most aspects of) a database schema.
- Schema Construct:
 - A ***component*** of the schema or an object within the schema, e.g., STUDENT, COURSE.

Schemas versus Instances

- Database State:
 - The actual data stored in a database at a ***particular moment in time***. This includes the collection of all the data in the database.
 - Also called database instance (or occurrence or snapshot).
 - The term *instance* is also applied to individual database components, e.g. *record instance*, *table instance*, *entity instance*

Database Schema vs. Database State

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

Database Schema vs. Database State (continued)

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

Example of a Database Schema

STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

GRADE_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

Figure 2.1

Schema diagram for the database in Figure 1.2.

Example of a database state

COURSE

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

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

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

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2

A database that stores student and course information.

Three-Schema Architecture

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

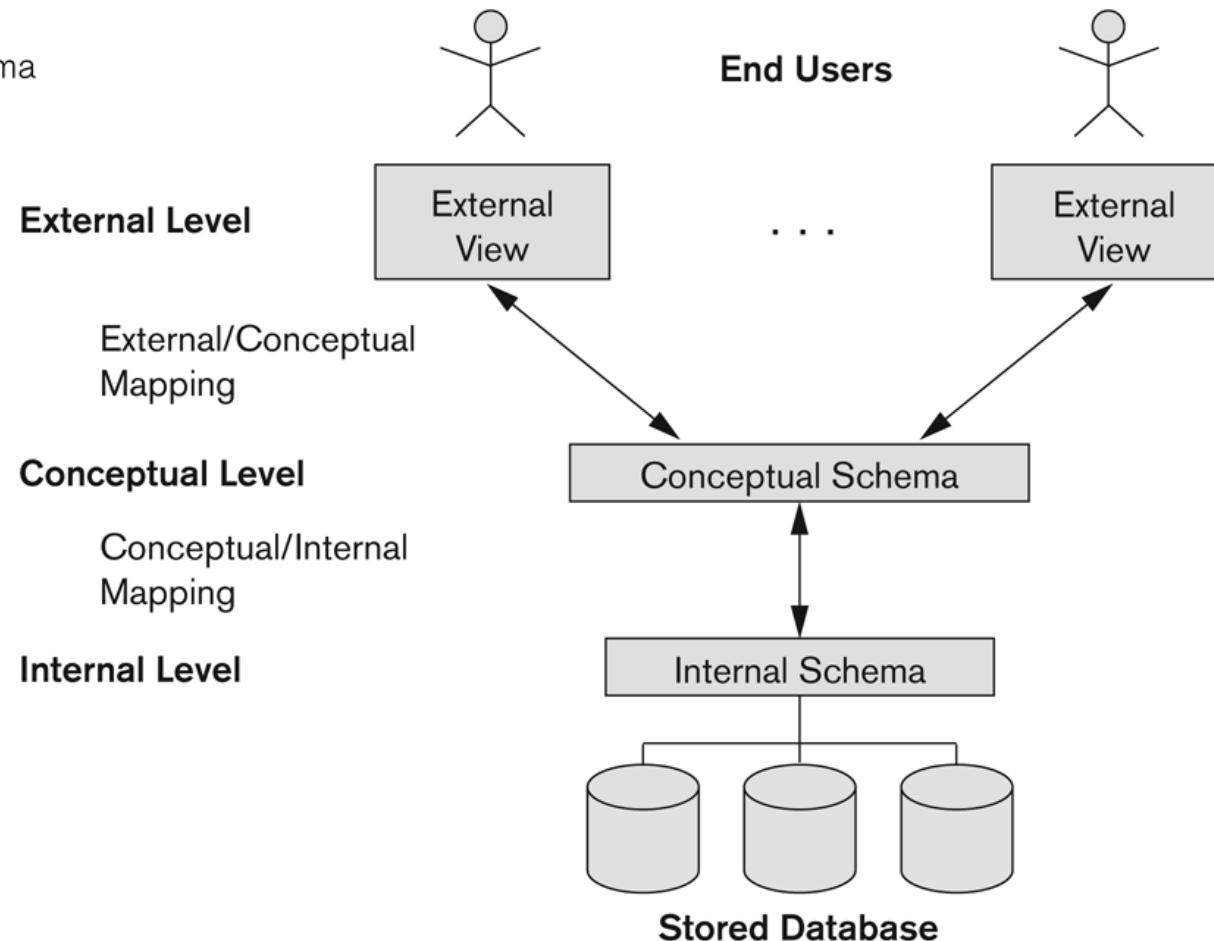
Three-Schema Architecture

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

The three-schema architecture

Figure 2.2

The three-schema architecture.



Three-Schema Architecture

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

Data Independence

- **Logical Data Independence:**
 - The capacity to change the conceptual schema without having to change the external schemas and their associated application programs.
- **Physical Data Independence:**
 - The capacity to change the internal schema without having to change the conceptual schema.
 - For example, the internal schema may be changed when certain file structures are reorganized or new indexes are created to improve database performance

Data Independence (continued)

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

DBMS Languages

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

DBMS Languages

■ Data Definition Language (DDL):

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

DBMS Languages

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

Types of DML

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

DBMS Interfaces

- Stand-alone query language interfaces
 - Example: Entering SQL queries at the DBMS interactive SQL interface (e.g. SQL*Plus in ORACLE)
- Programmer interfaces for embedding DML in programming languages
- User-friendly interfaces
 - Menu-based, forms-based, graphics-based, etc.
- Mobile Interfaces:interfaces allowing users to perform transactions using mobile apps

DBMS Programming Language Interfaces

- Programmer interfaces for embedding DML in a programming languages:
 - **Embedded Approach:** e.g embedded SQL (for C, C++, etc.), SQLJ (for Java)
 - **Procedure Call Approach:** e.g. JDBC for Java, ODBC (Open Database Connectivity) for other programming languages as API's (application programming interfaces)
 - **Database Programming Language Approach:** e.g. ORACLE has PL/SQL, a programming language based on SQL; language incorporates SQL and its data types as integral components
 - **Scripting Languages:** PHP (client-side scripting) and Python (server-side scripting) are used to write database programs.

User-Friendly DBMS Interfaces

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

Other DBMS Interfaces

- Natural language: free text as a query
- Speech : Input query and Output response
- Web Browser with keyword search
- Parametric interfaces, e.g., bank tellers using function keys.
- Interfaces for the DBA:
 - Creating user accounts, granting authorizations
 - Setting system parameters
 - Changing schemas or access paths

Database System Utilities

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

Other Tools

- Application Development Environments and CASE (computer-aided software engineering) tools:
- Examples:
 - PowerBuilder (Sybase)
 - JBuilder (Borland)
 - JDeveloper 10G (Oracle)

Typical DBMS Component Modules

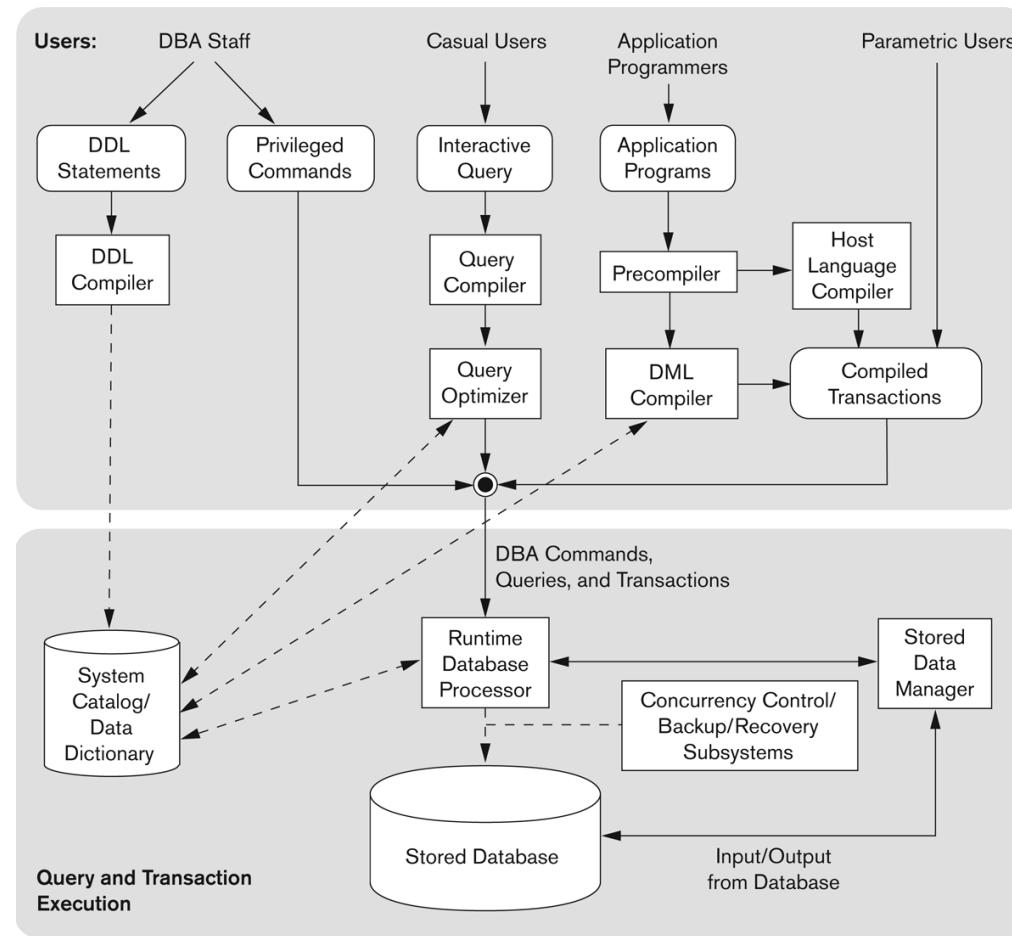


Figure 2.3
Component modules of a DBMS and their interactions.

Centralized and Client-Server DBMS Architectures

■ Centralized DBMS:

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

A Physical Centralized Architecture

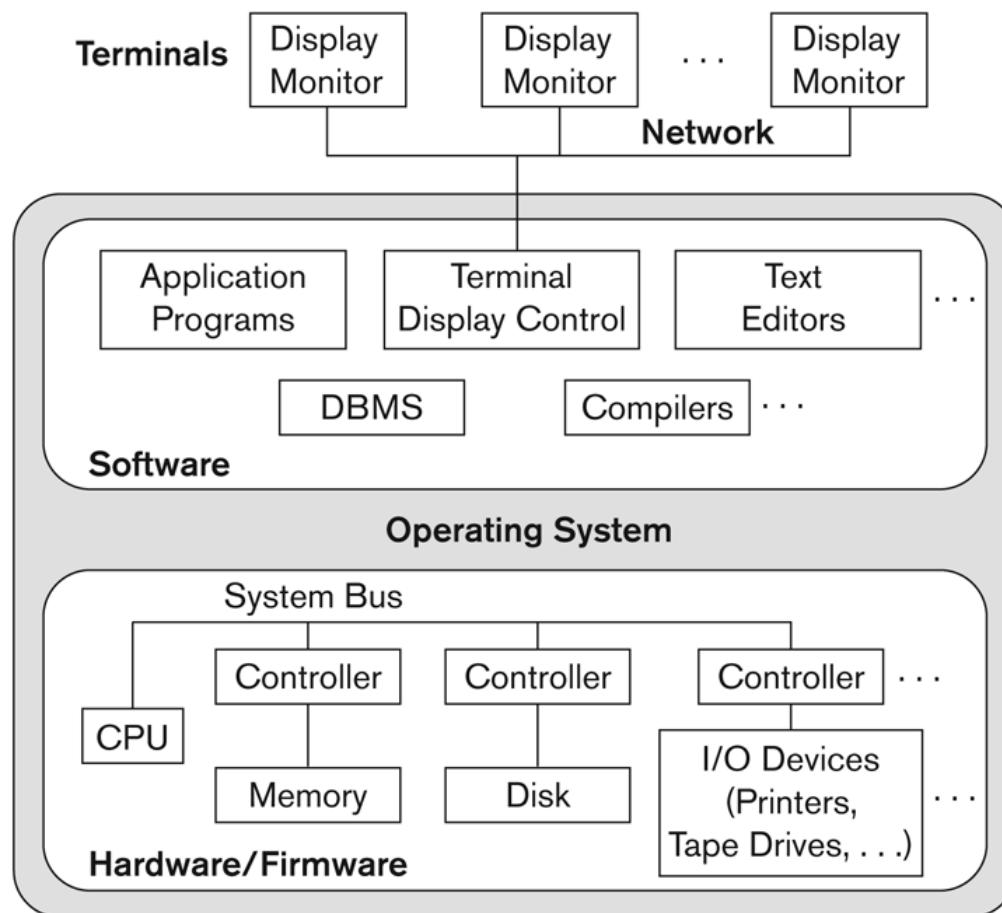


Figure 2.4

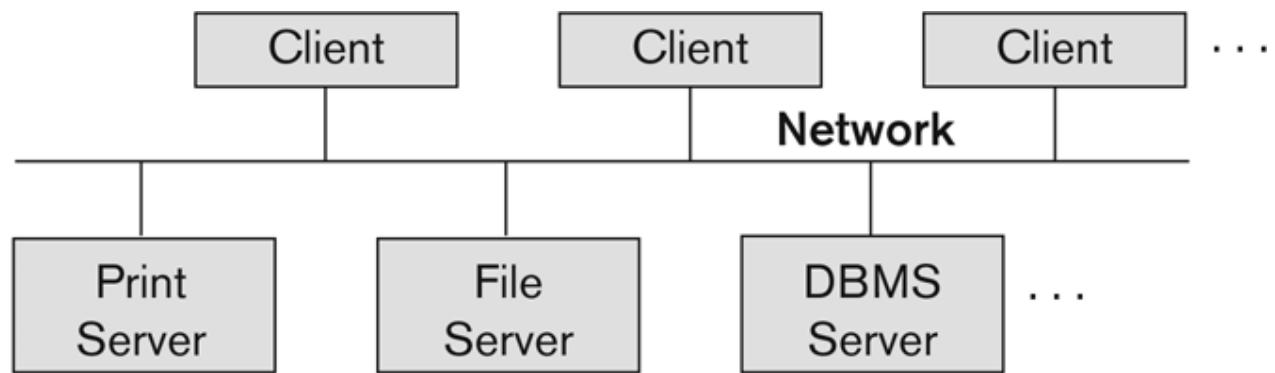
A physical centralized architecture.

Basic 2-tier Client-Server Architectures

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

Logical two-tier client server architecture

Figure 2.5
Logical two-tier
client/server
architecture.



Clients

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

DBMS Server

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

Two Tier Client-Server Architecture

- Client and server must install appropriate client module and server module software for ODBC or JDBC
- A client program may connect to several DBMSs, sometimes called the data sources.
- In general, data sources can be files or other non-DBMS software that manages data.
- See Chapter 10 for details on Database Programming

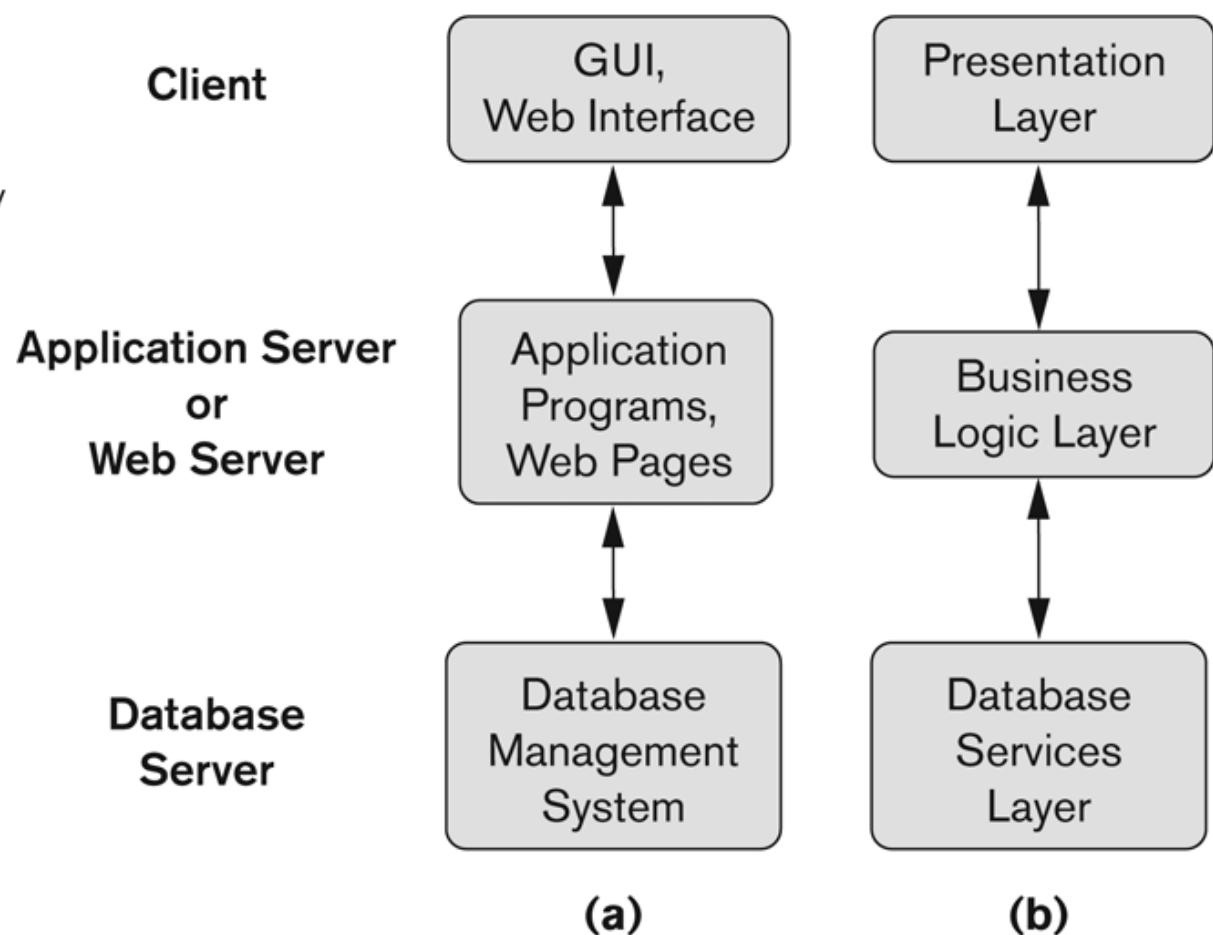
Three Tier Client-Server Architecture

- Common for Web applications
- Intermediate Layer called Application Server or Web Server:
 - Stores the web connectivity software and the business logic part of the application used to access the corresponding data from the database server
 - Acts like a conduit for sending partially processed data between the database server and the client.
- Three-tier Architecture Can Enhance Security:
 - Database server only accessible via middle tier
 - Clients cannot directly access database server
 - Clients contain user interfaces and Web browsers
 - The client is typically a PC or a mobile device connected to the Web

Three-tier client-server architecture

Figure 2.7

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



Classification of DBMSs

- Based on the data model used
 - Legacy: Network, Hierarchical.
 - Currently Used: Relational, Object-oriented, Object-relational
 - Recent Technologies: Key-value storage systems, NOSQL systems: document based, column-based, graph-based and key-value based. Native XML DBMSs.
- Other classifications
 - Single-user (typically used with personal computers) vs. multi-user (most DBMSs).
 - Centralized (uses a single computer with one database) vs. distributed (multiple computers, multiple DBs)

Variations of Distributed DBMSs (DDBMSs)

- Homogeneous DDBMS
- Heterogeneous DDBMS
- Federated or Multidatabase Systems
 - Participating Databases are loosely coupled with high degree of autonomy.
- 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.

Cost considerations for DBMSs

- Cost Range: from free open-source systems to configurations costing millions of dollars
- Examples of free relational DBMSs: MySQL, PostgreSQL, others
- Commercial DBMS offer additional specialized modules, e.g. time-series module, spatial data module, document module, XML module
 - These offer additional specialized functionality when purchased separately
 - Sometimes called cartridges (e.g., in Oracle) or blades
- Different licensing options: site license, maximum number of concurrent users (seat license), single user, etc.

Other Considerations

- Type of access paths within database system
 - E.g.- inverted indexing based (ADABAS is one such system). Fully indexed databases provide access by any keyword (used in search engines)
- General Purpose vs. Special Purpose
 - E.g.- Airline Reservation systems or many others-reservation systems for hotel/car etc. Are special purpose OLTP (Online Transaction Processing Systems)

History of Data Models (Additional Material)

- Network Model
- Hierarchical Model
- Relational Model
- Object-oriented Data Models
- Object-Relational Models

History of Data Models

■ Network Model:

- The first network DBMS was implemented by Honeywell in 1964-65 (IDS System).
- Adopted heavily due to the support by CODASYL (Conference on Data Systems Languages) (CODASYL - DBTG report of 1971).
- Later implemented in a large variety of systems - IDMS (Cullinet - now Computer Associates), DMS 1100 (Unisys), IMAGE (H.P. (Hewlett-Packard)), VAX -DBMS (Digital Equipment Corp., next COMPAQ, now H.P.).

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.

Network Model

- 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”

History of Data Models

■ Hierarchical Data Model:

- Initially implemented in a joint effort by IBM and North American Rockwell around 1965. Resulted in the IMS family of systems.
- IBM's IMS product had (and still has) a very large customer base worldwide
- Hierarchical model was formalized based on the IMS system
- Other systems based on this model: System 2k (SAS inc.)

Hierarchical Model

- Advantages:
 - Simple to construct and operate
 - Corresponds to a number of natural hierarchically organized domains, e.g., organization ("org") chart
 - 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"

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 (e.g. DB2, ORACLE, MS SQL Server, SYBASE, INFORMIX).
- Several free open source implementations, e.g. MySQL, PostgreSQL
- Currently most dominant for developing database applications.
- SQL relational standards: SQL-89 (SQL1), SQL-92 (SQL2), SQL-99, SQL3, ...
- Chapters 5 through 11 describe this model in detail

History of Data Models

■ Object-oriented Data Models:

- 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 Database Standard: ODMG-93, ODMG-version 2.0, ODMG-version 3.0.
- Chapter 12 describes this model.

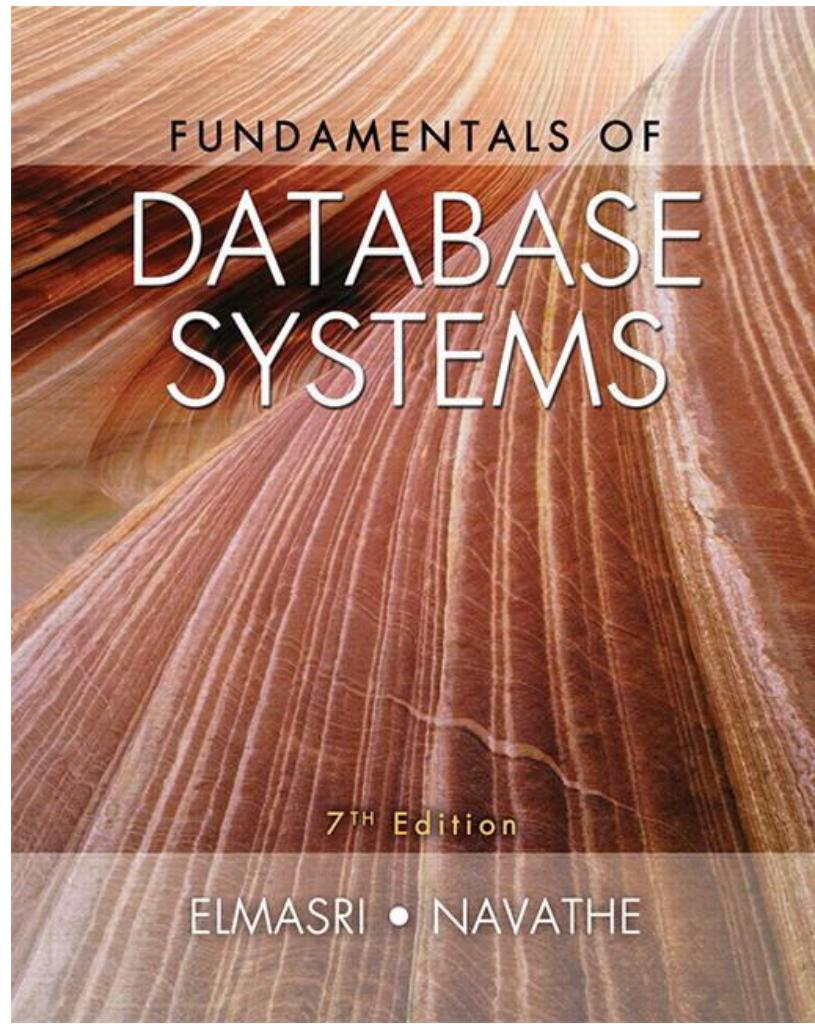
History of Data Models

■ Object-Relational Models:

- The trend to mix object models with relational was started with Informix Universal Server.
- Relational systems incorporated concepts from object databases leading to object-relational.
- Exemplified in the versions of Oracle, DB2, and SQL Server and other DBMSs.
- Current trend by Relational DBMS vendors is to extend relational DBMSs with capability to process XML, Text and other data types.
- The term “Object-relational” is receding in the marketplace.

Chapter Summary

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



CHAPTER 3

Data Modeling Using the Entity-Relationship (ER) Model

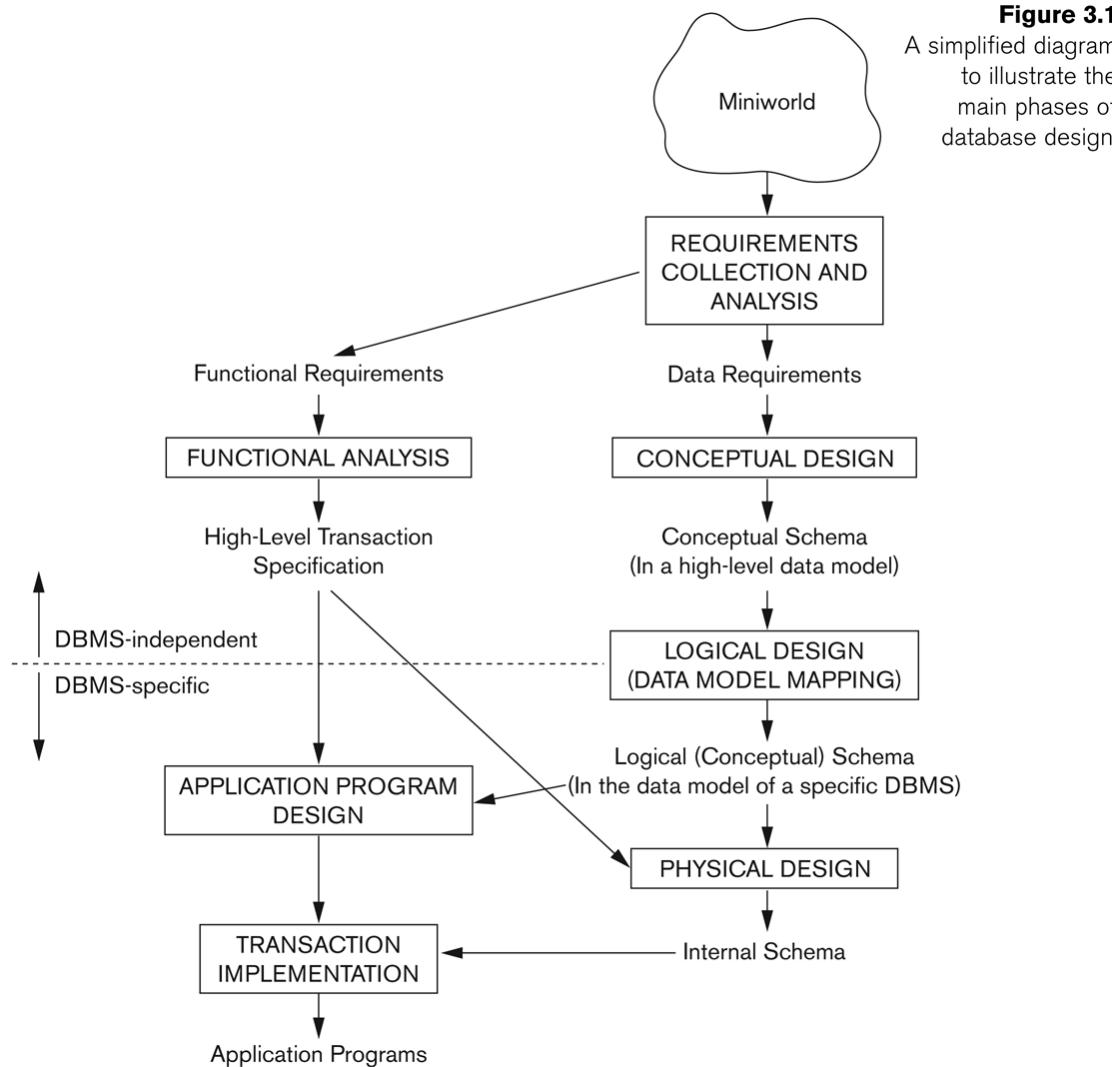
Chapter Outline

- Overview of Database Design Process
- 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
- Relationships of Higher Degree

Overview of Database Design Process

- Two main activities:
 - Database design
 - Applications design
- Focus in this chapter on conceptual database design
 - To design the conceptual schema for a database application
- Applications design focuses on the programs and interfaces that access the database
 - Generally considered part of software engineering

Overview of Database Design Process



Methodologies for Conceptual Design

- Entity Relationship (ER) Diagrams (This Chapter)
- Enhanced Entity Relationship (EER) Diagrams (Chapter 4)
- Use of Design Tools in industry for designing and documenting large scale designs
- The UML (Unified Modeling Language) Class Diagrams are popular in industry to document conceptual database designs

Example COMPANY Database

- We need to create a database schema design based on the following (simplified) **requirements** of the COMPANY Database:
 - 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. A department may have several locations.
 - Each department *controls* a number of PROJECTs. Each project has a unique name, unique number and is located at a single location.

Example COMPANY Database (Continued)

- The database will store each EMPLOYEE's social security number, address, salary, sex, and birthdate.
 - Each employee *works for* one department but may *work on* several projects.
 - The DB will keep track of the number of hours per week that an employee currently works on each project.
 - It is required to keep track of the *direct supervisor* of each employee.
- Each employee may *have* a number of DEPENDENTS.
 - For each dependent, the DB keeps a record of name, **sex, birthdate, and relationship to the employee.**

ER Model Concepts

- Entities and Attributes
 - Entity is a basic concept for the ER model. Entities are specific things or objects 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 the attributes 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, date, 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)}
 - Multiple PreviousDegrees values can exist
 - Each has four subcomponent attributes:
 - College, Year, Degree, Field

Example of a composite attribute

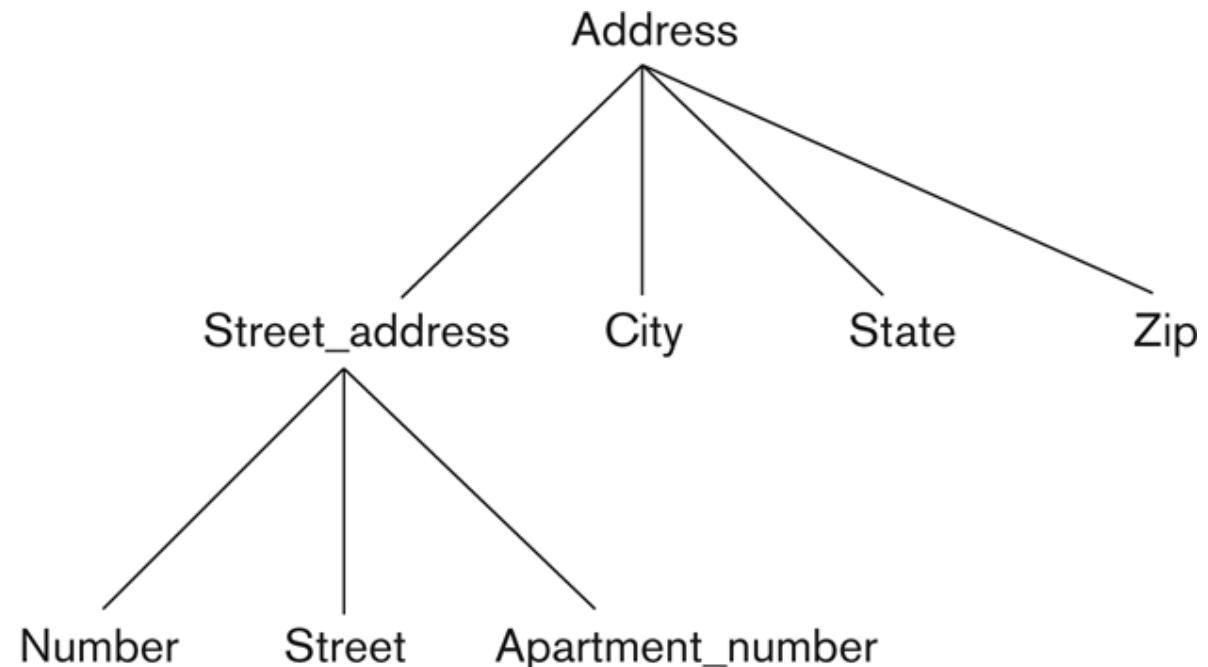


Figure 3.4

A hierarchy of composite attributes.

Entity Types and Key Attributes (1)

- Entities with the same basic attributes are grouped or typed into an entity type.
 - For example, the entity type **EMPLOYEE** and **PROJECT**.
- 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**.

Entity Types and Key Attributes (2)

- A key attribute may be composite.
 - VehicleTagNumber is a key of the CAR entity type with components (Number, State).
- An entity type may have more than one key.
 - The CAR entity type may have two keys:
 - VehicleIdentificationNumber (popularly called VIN)
 - VehicleTagNumber (Number, State), aka license plate number.
- Each key is underlined (Note: this is different from the relational schema where only one “primary key is underlined”).

Entity Set

- Each entity type will have a collection of entities stored in the database
 - Called the **entity set** or sometimes **entity collection**
- Previous slide shows three CAR entity instances in the entity set for CAR
- Same name (CAR) used to refer to both the entity type and the entity set
- However, entity type and entity set may be given different names
- Entity set is the current *state* of the entities of that type that are stored in the database

Value Sets (Domains) of Attributes

- Each simple attribute is associated with a value set
 - E.g., Lastname has a value which is a character string of upto 15 characters, say
 - Date has a value consisting of MM-DD-YYYY where each letter is an integer
- A **value set** specifies the set of values associated with an attribute

Attributes and Value Sets

- Value sets are similar to data types in most programming languages – e.g., integer, character (n), real, bit
- Mathematically, an attribute A for an entity type E whose value set is V is defined as a function

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

Where $P(V)$ indicates a power set (which means all possible subsets) of V. The above definition covers simple and multivalued attributes.

- We refer to the value of attribute A for entity e as $A(e)$.

Displaying an Entity type

- In ER diagrams, an entity type is displayed in a rectangular box
- Attributes are displayed in ovals
 - Each attribute is connected to its entity type
 - Components of a composite attribute are connected to the oval representing the composite attribute
 - Each key attribute is underlined
 - Multivalued attributes displayed in double ovals
- See the full ER notation in advance on the next slide

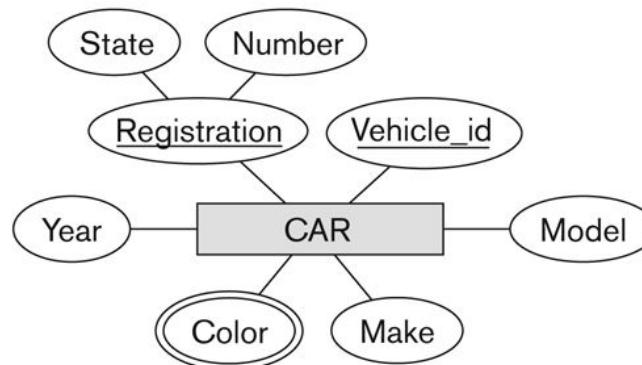
NOTATION for ER diagrams

Figure 3.14
Summary of the
notation for ER
diagrams.

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

Entity Type CAR with two keys and a corresponding Entity Set

(a)



(b)

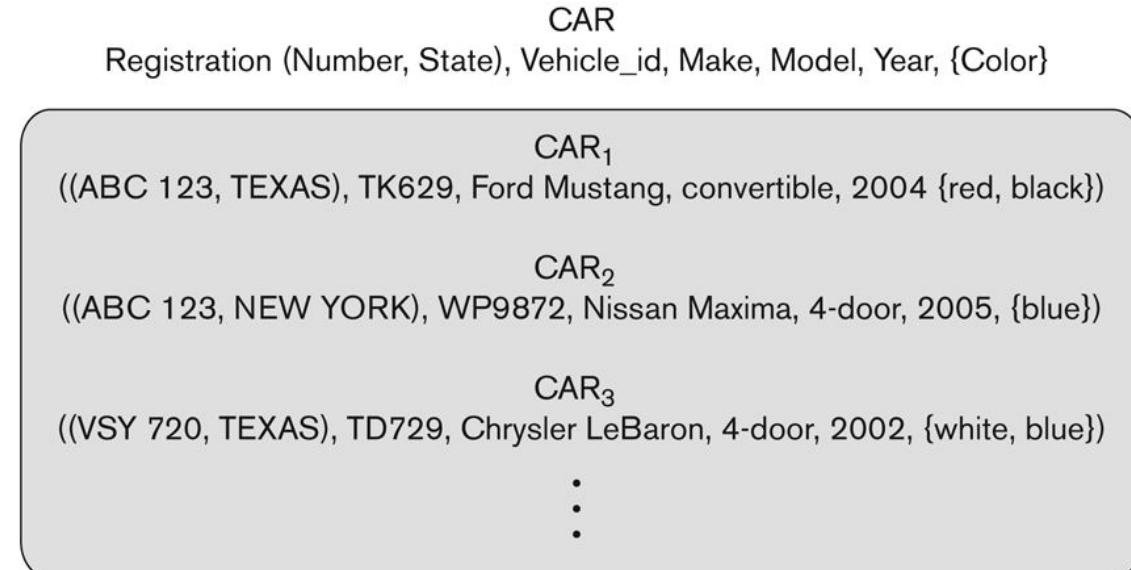


Figure 3.7

The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

Initial Conceptual Design of Entity Types for the COMPANY Database Schema

- Based on the requirements, we can identify four initial entity types in the COMPANY database:
 - DEPARTMENT
 - PROJECT
 - EMPLOYEE
 - DEPENDENT
- Their initial conceptual design is shown on the following slide
- The initial attributes shown are derived from the requirements description

Initial Design of Entity Types: EMPLOYEE, DEPARTMENT, PROJECT, DEPENDENT

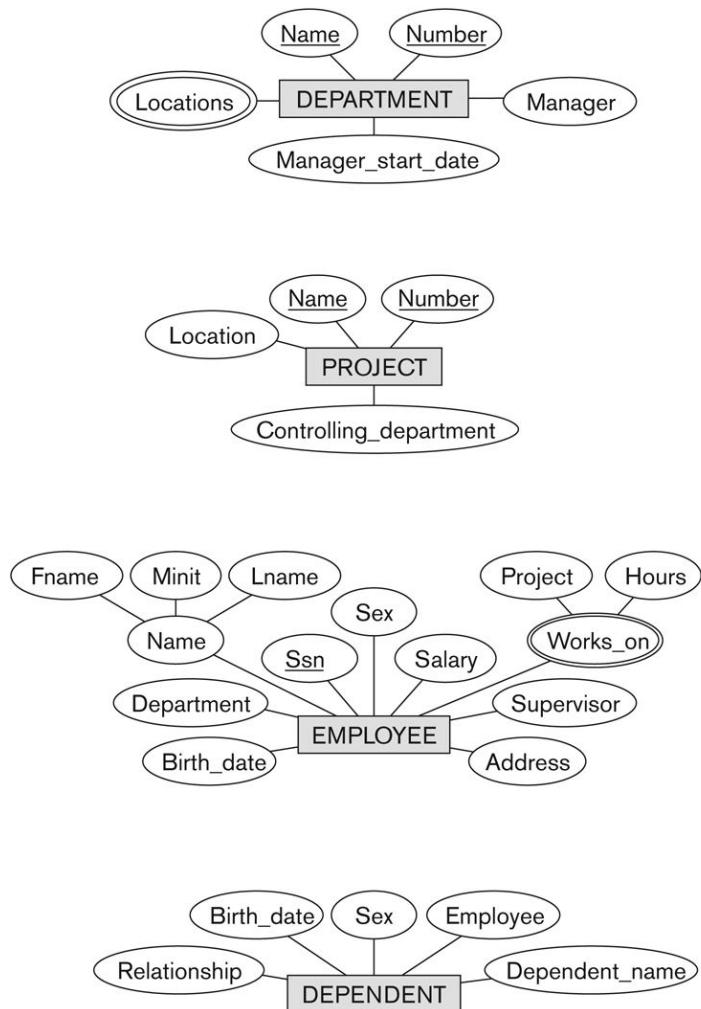


Figure 3.8

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

Refining the initial design by introducing relationships

- The initial design is typically not complete
- Some aspects in the requirements will be represented as **relationships**
- ER model has three main concepts:
 - Entities (and their entity types and entity sets)
 - Attributes (simple, composite, multivalued)
 - Relationships (and their relationship types and relationship sets)
- We introduce relationship concepts next

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.

Relationship instances of the WORKS_FOR N:1 relationship between EMPLOYEE and DEPARTMENT

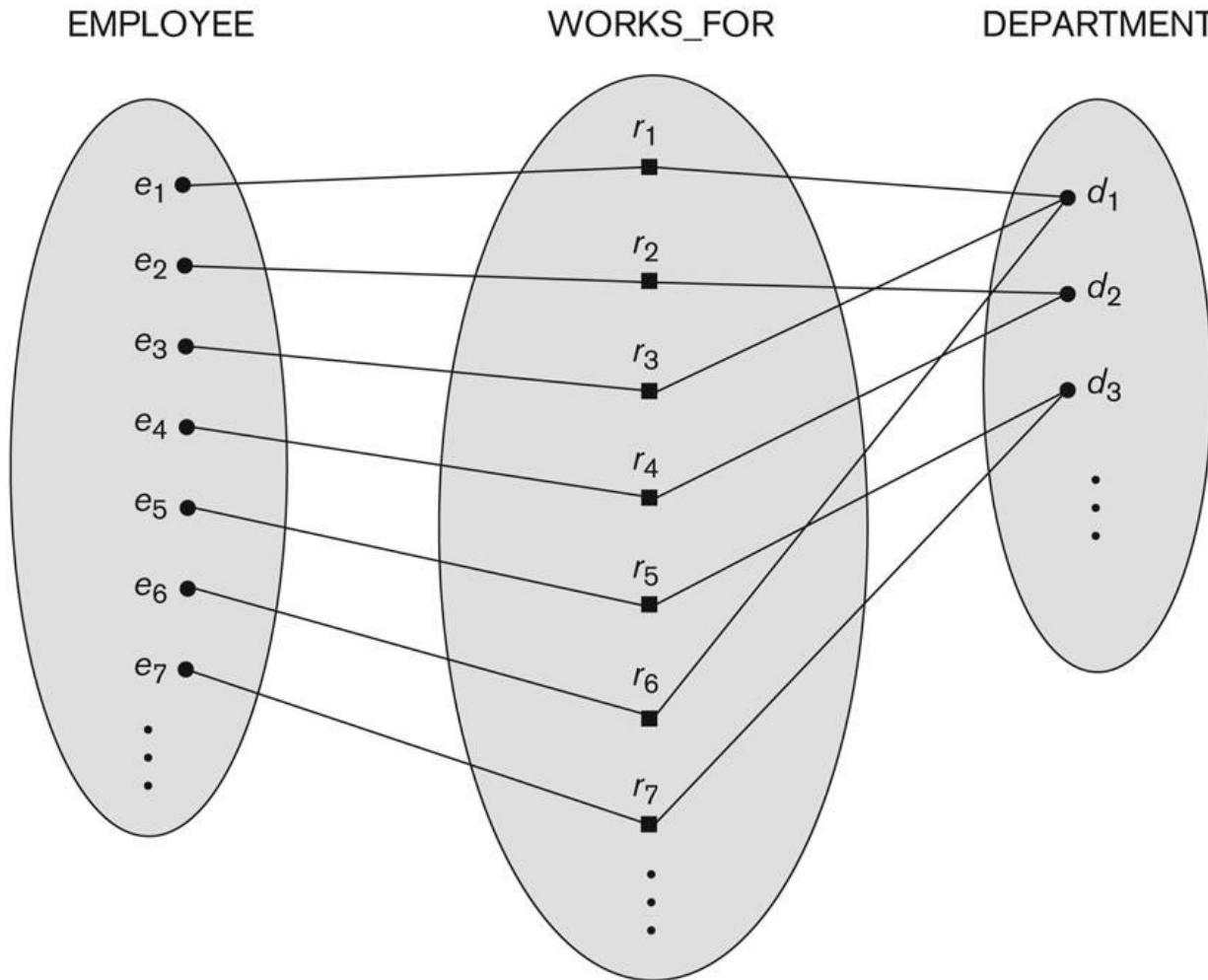


Figure 3.9

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

Relationship instances of the M:N WORKS_ON relationship between EMPLOYEE and PROJECT

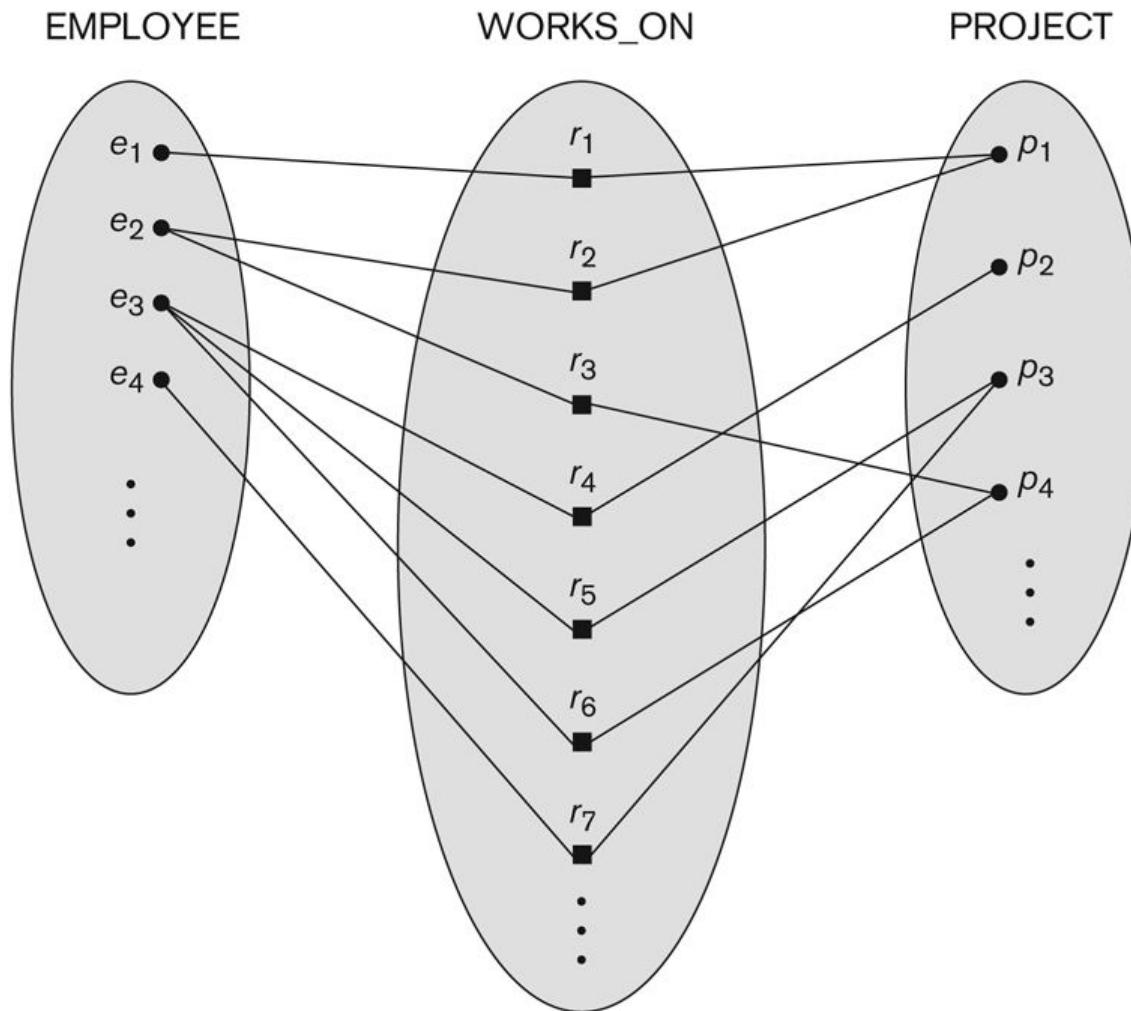


Figure 3.13
An M:N relationship,
WORKS_ON.

Relationship type vs. relationship set (1)

- Relationship Type:
 - Is the schema description of a relationship
 - Identifies the relationship name and the participating entity types
 - Also identifies certain relationship constraints
- Relationship Set:
 - The current set of relationship instances represented in the database
 - The current *state* of a relationship type

Relationship type vs. relationship set (2)

- Previous figures displayed the relationship sets
- Each instance in the set relates individual participating entities – one from each participating entity type
- In ER diagrams, we represent the *relationship type* as follows:
 - Diamond-shaped box is used to display a relationship type
 - Connected to the participating entity types via straight lines
 - Note that the relationship type is not shown with an arrow. The name should be typically be readable from left to right and top to bottom.

Refining the COMPANY database schema by introducing relationships

- By examining the requirements, six relationship types are identified
- All are *binary* relationships(degree 2)
- Listed below with their participating entity types:
 - WORKS_FOR (between EMPLOYEE, DEPARTMENT)
 - MANAGES (also between EMPLOYEE, DEPARTMENT)
 - CONTROLS (between DEPARTMENT, PROJECT)
 - WORKS_ON (between EMPLOYEE, PROJECT)
 - SUPERVISION (between EMPLOYEE (as subordinate), EMPLOYEE (as supervisor))
 - DEPENDENTS_OF (between EMPLOYEE, DEPENDENT)

ER DIAGRAM – Relationship Types are:

WORKS_FOR, MANAGES, WORKS_ON, CONTROLS, SUPERVISION, DEPENDENTS_OF

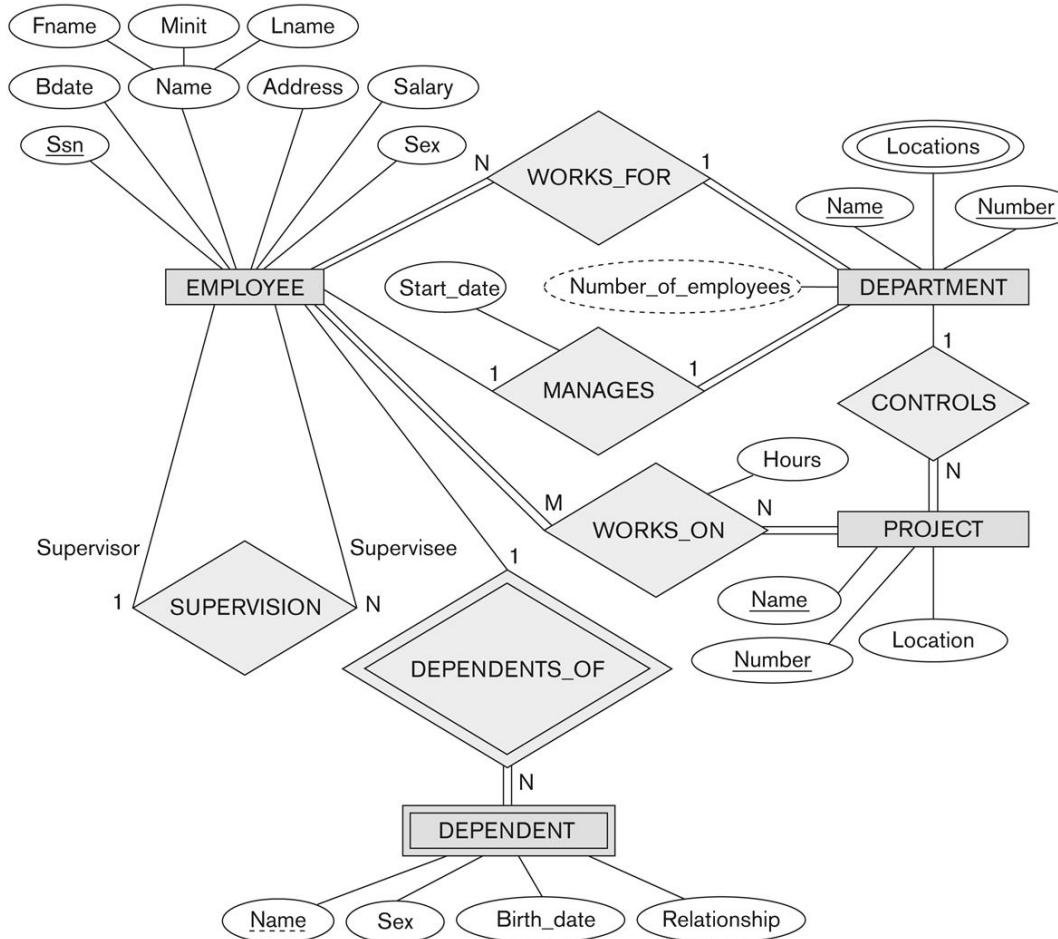


Figure 3.2

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Discussion on Relationship Types

- In the refined design, some attributes from the initial entity types are refined into relationships:
 - Manager of DEPARTMENT -> MANAGES
 - Works_on of EMPLOYEE -> WORKS_ON
 - Department of EMPLOYEE -> WORKS_FOR
 - etc
- In general, more than one relationship type can exist between the same participating entity types
 - MANAGES and WORKS_FOR are distinct relationship types between EMPLOYEE and DEPARTMENT
 - Different meanings and different relationship instances.

Constraints on Relationships

- Constraints on Relationship Types
 - (Also known as ratio constraints)
 - Cardinality Ratio (specifies *maximum* participation)
 - One-to-one (1:1)
 - One-to-many (1:N) or Many-to-one (N:1)
 - Many-to-many (M:N)
 - Existence Dependency Constraint (specifies *minimum* participation) (also called participation constraint)
 - zero (optional participation, not existence-dependent)
 - one or more (mandatory participation, existence-dependent)

Many-to-one (N:1) Relationship

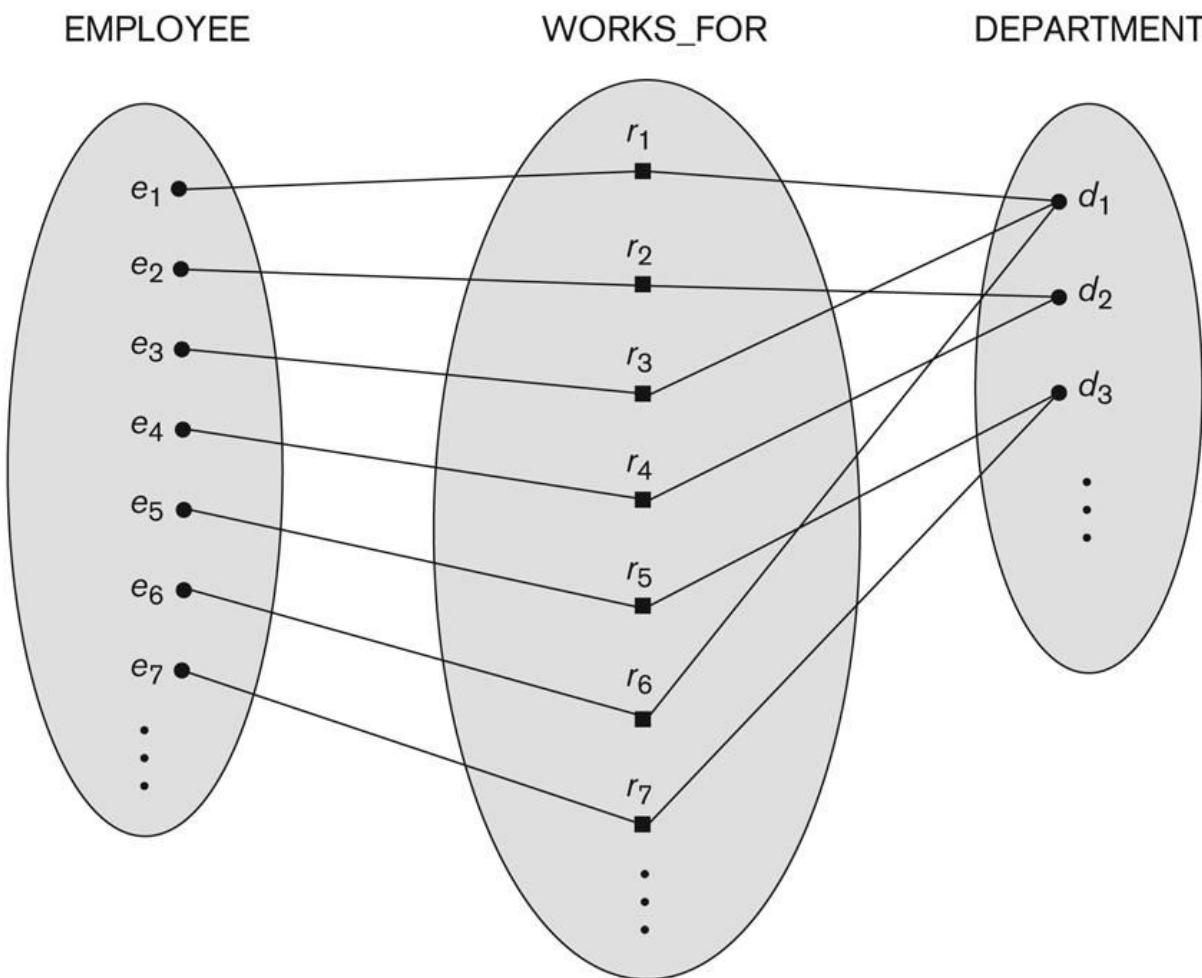


Figure 3.9

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

Many-to-many (M:N) Relationship

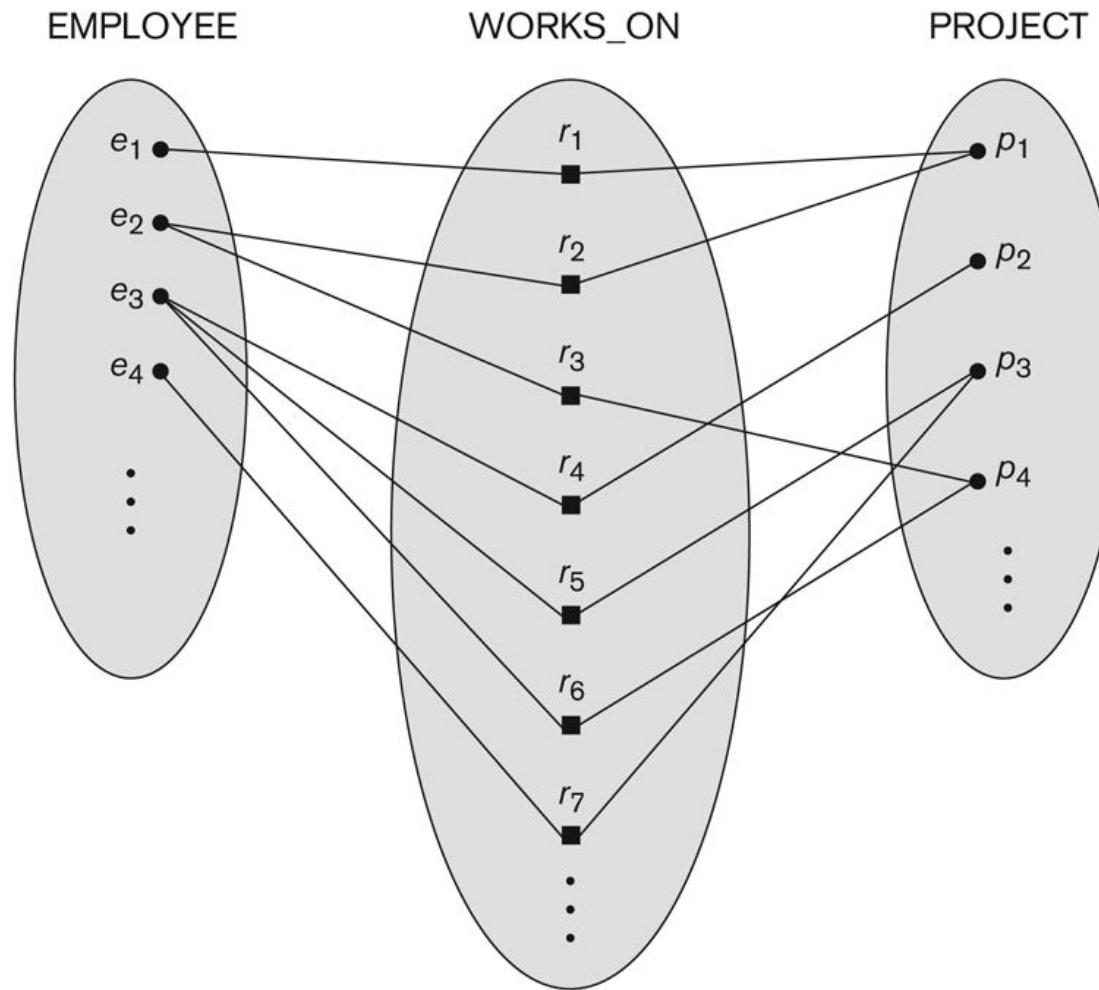


Figure 3.13
An M:N relationship,
WORKS_ON.

Recursive Relationship Type

- A relationship type between the same participating entity type in **distinct roles**
- Also called a **self-referencing** relationship type.
- Example: the SUPERVISION relationship
- EMPLOYEE participates twice in two distinct roles:
 - supervisor (or boss) role
 - supervisee (or subordinate) role
- Each relationship instance relates two distinct EMPLOYEE entities:
 - One employee in *supervisor* role
 - One employee in *supervisee* role

Displaying a recursive relationship

- In 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`

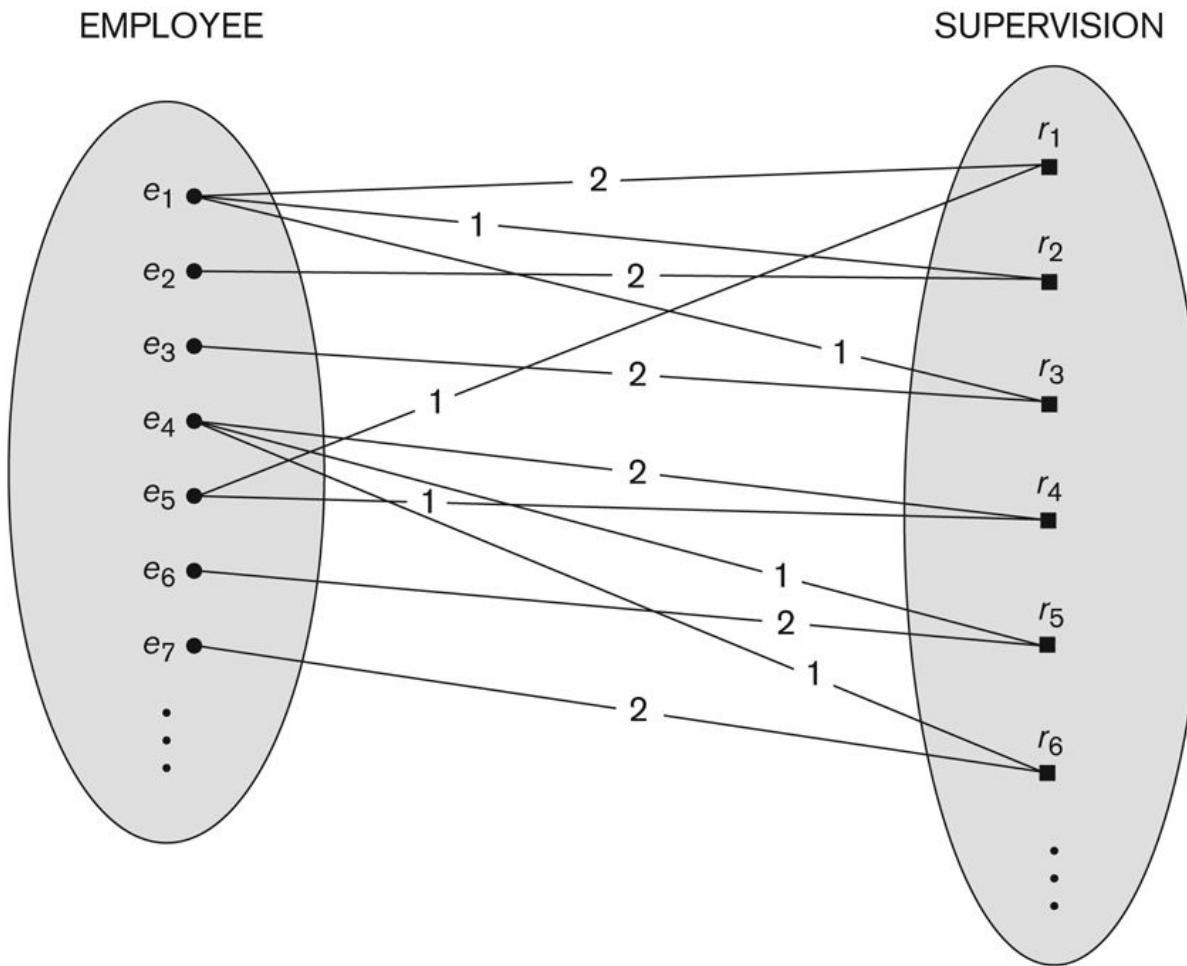


Figure 3.11

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

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

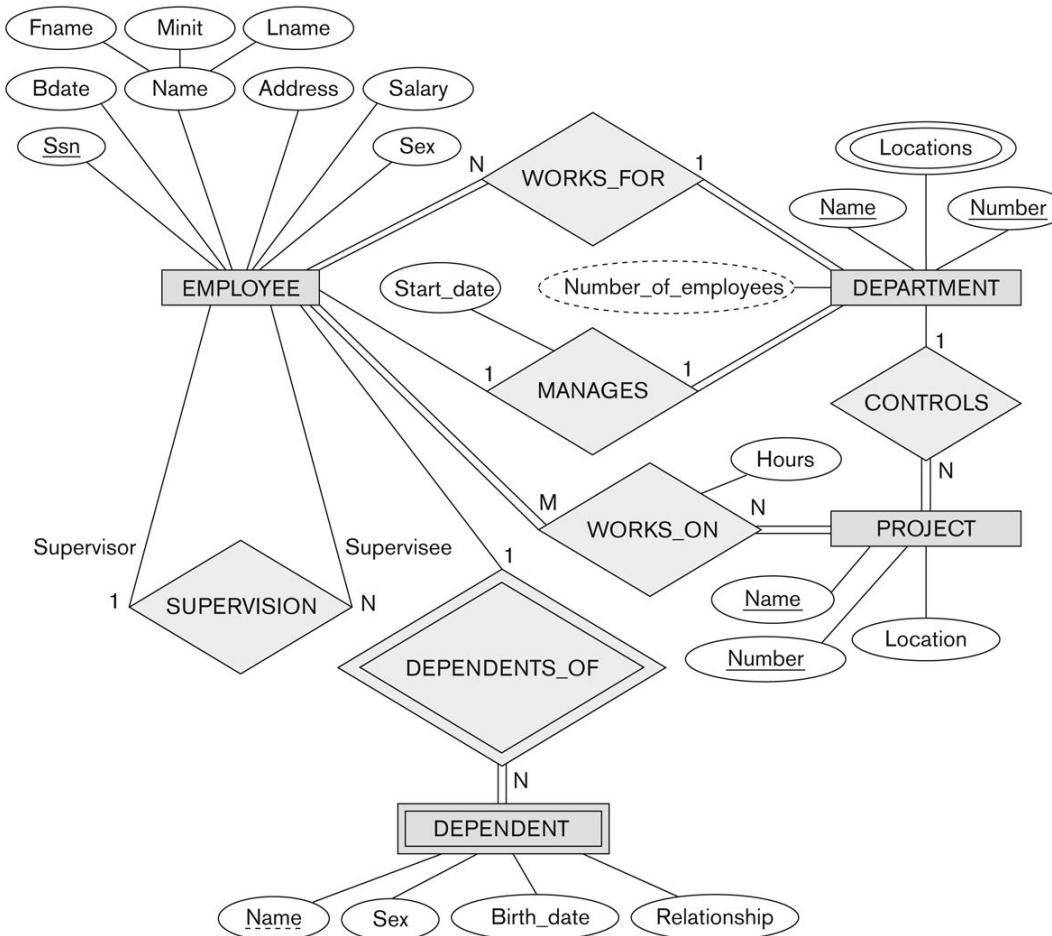


Figure 3.2

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Weak Entity Types

- An entity that does not have a key attribute and that is identification-dependent on another entity type.
- 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 relationship type
- **Example:**
 - A DEPENDENT entity is identified by the dependent's first name, *and* the specific EMPLOYEE with whom the dependent is related
 - Name of DEPENDENT is the *partial key*
 - DEPENDENT is a *weak entity type*
 - EMPLOYEE is its identifying entity type via the identifying relationship type DEPENDENT_OF

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.
 - A value of HoursPerWeek depends on a particular (employee, project) combination
 - Most relationship attributes are used with M:N relationships
 - In 1:N relationships, they can be transferred to the entity type on the N-side of the relationship

Example Attribute of a Relationship Type: Hours of WORKS_ON

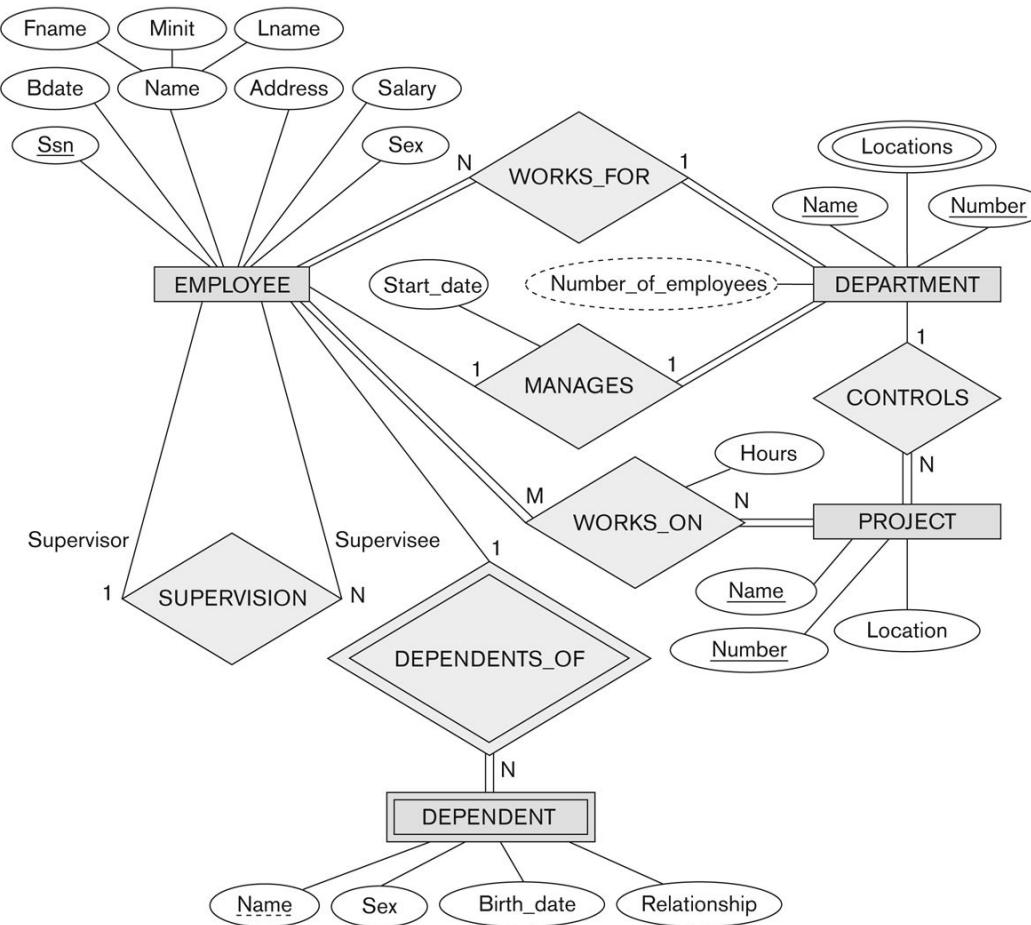


Figure 3.2

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

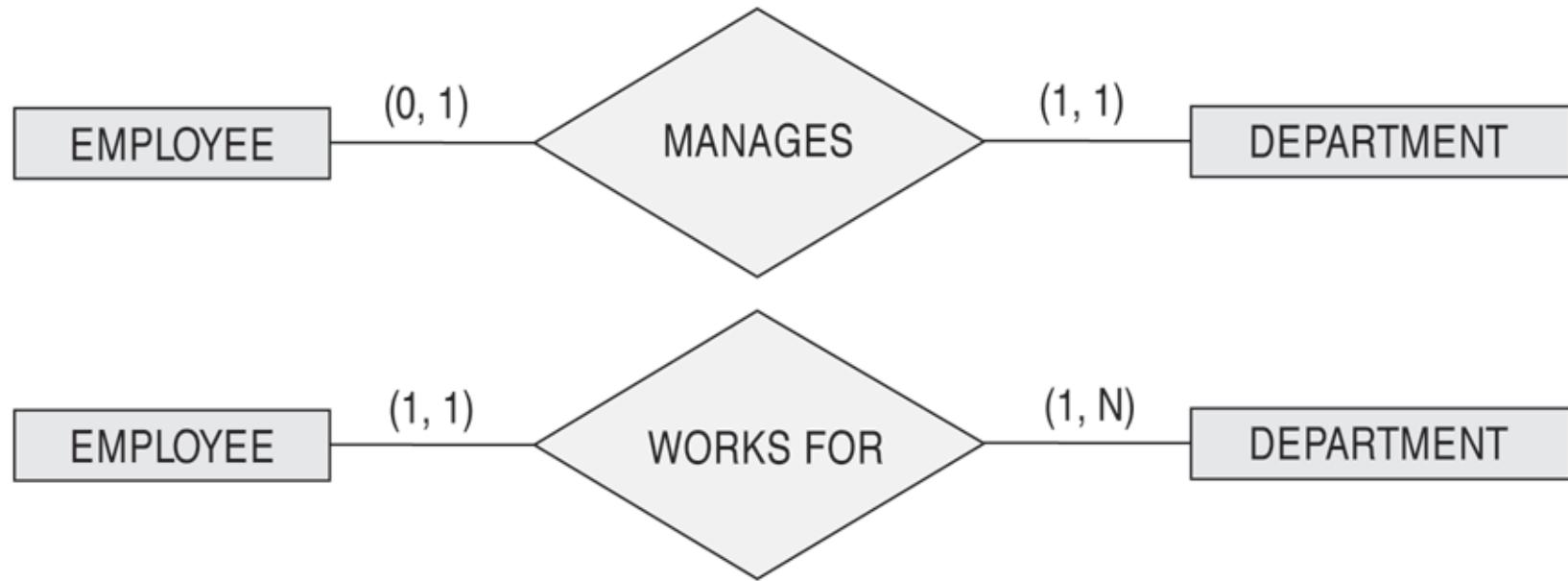
Notation for Constraints on Relationships

- Cardinality ratio (of a binary relationship): 1:1, 1:N, N:1, or M:N
 - Shown by placing appropriate numbers on the relationship edges.
- Participation constraint (on each participating entity type): total (called existence dependency) or partial.
 - Total shown by double line, partial by single line.
- 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 (signifying no limit)
- 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 for relationship constraints



Read the min,max numbers next to the entity type and looking **away from** the entity type

COMPANY ER Schema Diagram using (min, max) notation

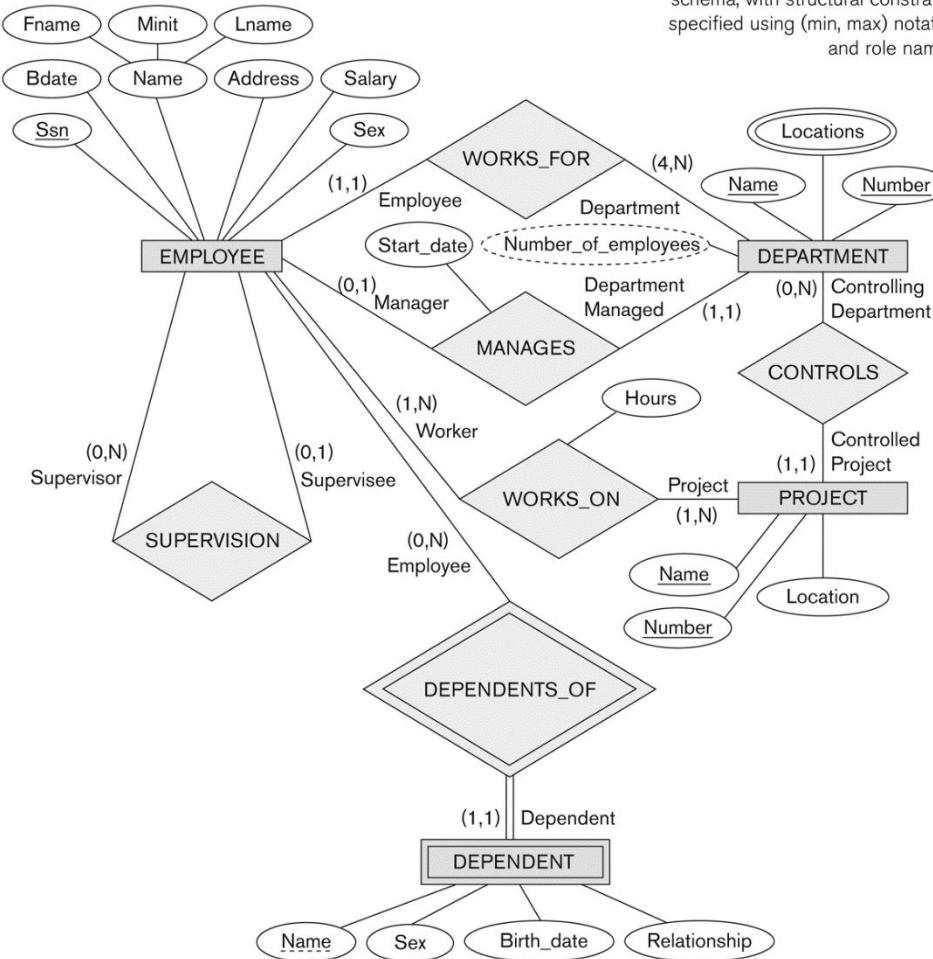


Figure 3.15

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

Alternative diagrammatic notation

- ER diagrams is one popular example for displaying database schemas
- Many other notations exist in the literature and in various database design and modeling tools
- Appendix A illustrates some of the alternative notations that have been used
- UML class diagrams is representative of another way of displaying ER concepts that is used in several commercial design tools

Summary of notation for ER diagrams

Figure 3.14
Summary of the
notation for ER
diagrams.

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

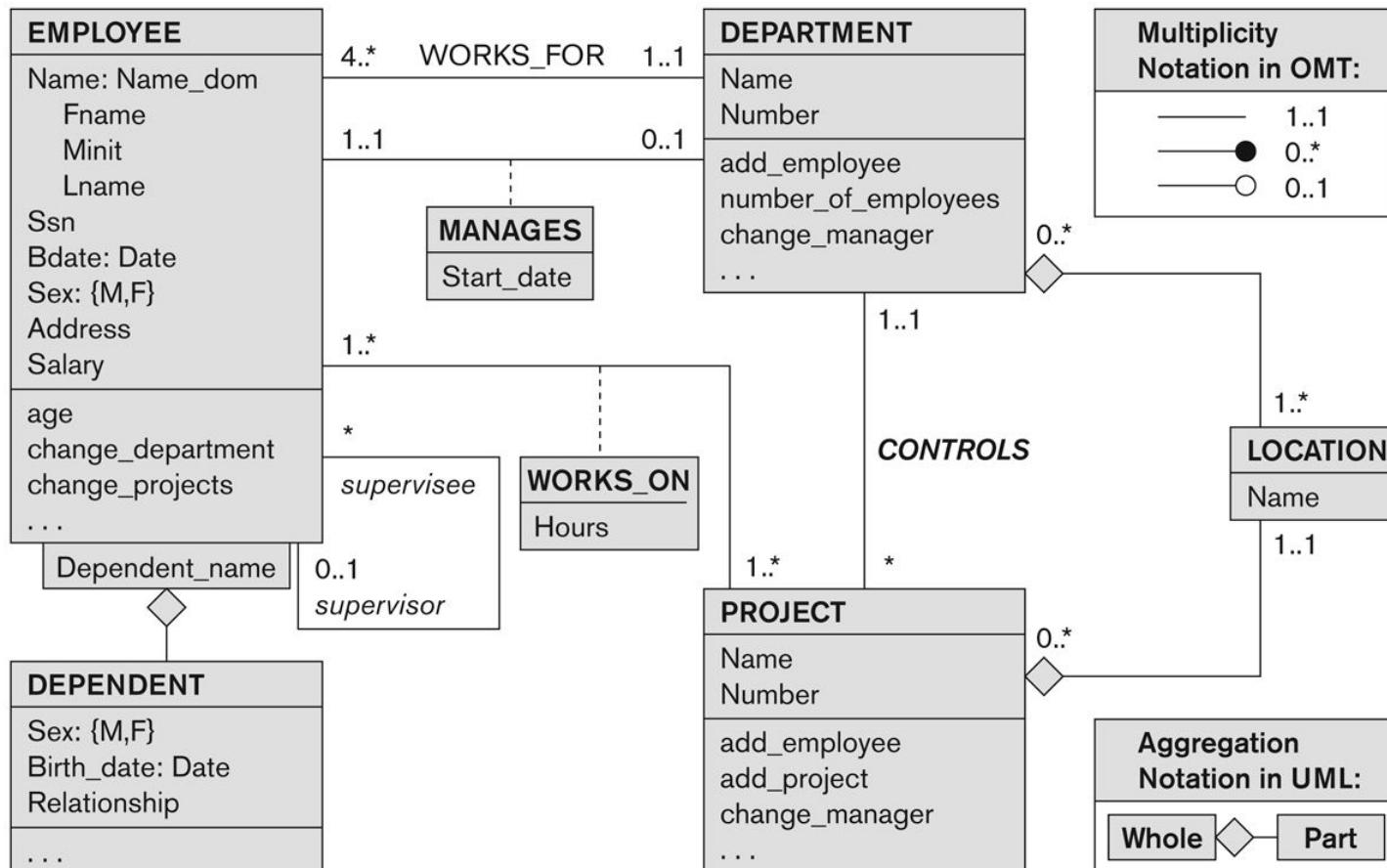
UML class diagrams

- Represent classes (similar to entity types) as large rounded boxes with three sections:
 - Top section includes entity type (class) name
 - Second section includes attributes
 - Third section includes class operations (operations are not in basic ER model)
- Relationships (called associations) represented as lines connecting the classes
 - Other UML terminology also differs from ER terminology
- Used in database design and object-oriented software design
- UML has many other types of diagrams for software design

UML class diagram for COMPANY database schema

Figure 3.16

The COMPANY conceptual schema in UML class diagram notation.



Other alternative diagrammatic notations

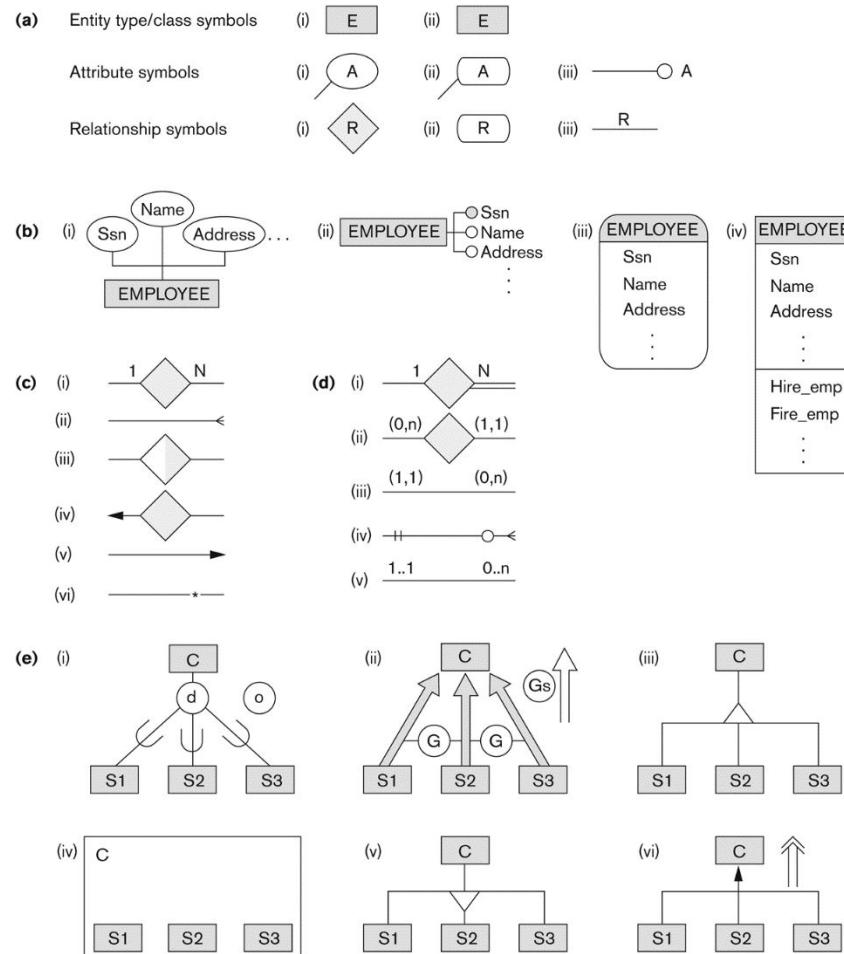


Figure A.1

Alternative notations. (a) Symbols for entity type/class, attribute, and relationship. (b) Displaying attributes. (c) Displaying cardinality ratios. (d) Various (min, max) notations. (e) Notations for displaying specialization/generalization.

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
- Constraints are harder to specify for higher-degree relationships ($n > 2$) than for binary relationships

Discussion of n-ary relationships ($n > 2$)

- In general, 3 binary relationships can represent different information than a single ternary relationship (see Figure 3.17a and b on next slide)
- If needed, the binary and n-ary relationships can all be included in the schema design (see Figure 3.17a and b, where all relationships convey different meanings)
- In some cases, a ternary relationship can be represented as a weak entity if the data model allows a weak entity type to have multiple identifying relationships (and hence multiple owner entity types) (see Figure 3.17c)

Example of a ternary relationship

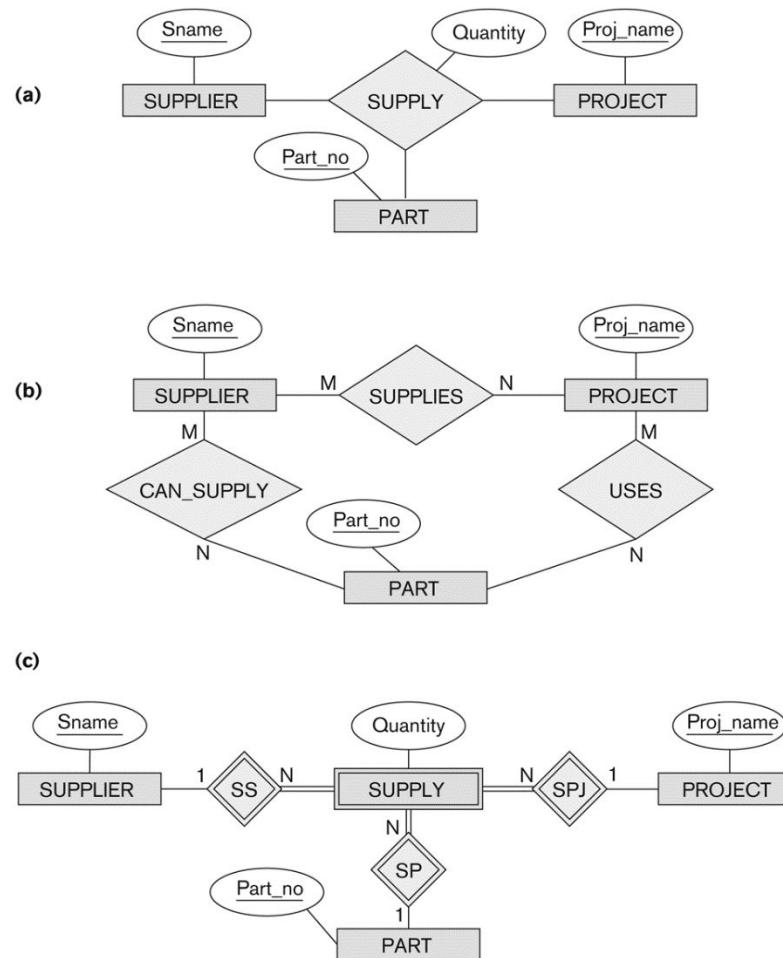


Figure 3.17

Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

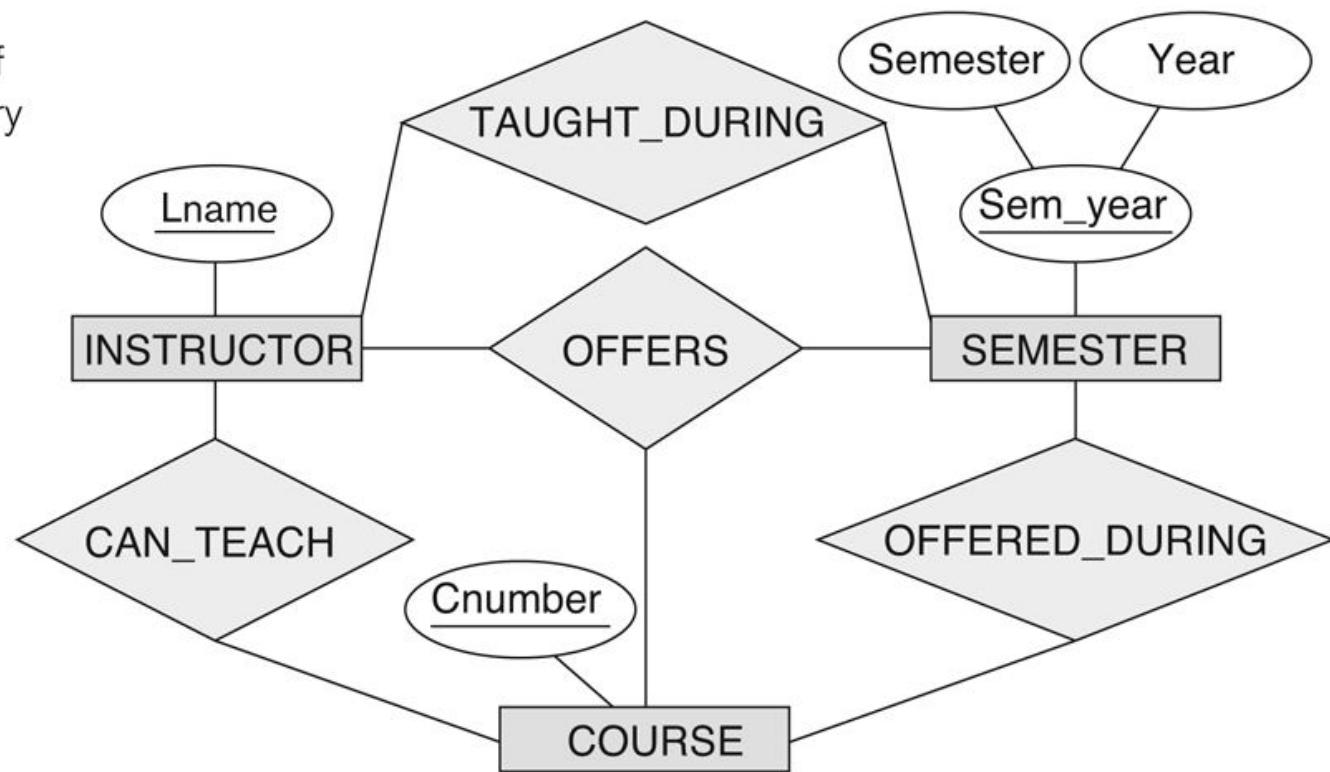
Discussion of n-ary relationships ($n > 2$)

- If a particular binary relationship can be derived from a higher-degree relationship at all times, then it is redundant
- For example, the TAUGHT_DURING binary relationship in Figure 3.18 (see next slide) can be derived from the ternary relationship OFFERS (based on the meaning of the relationships)

Another example of a ternary relationship

Figure 3.18

Another example of ternary versus binary relationship types.



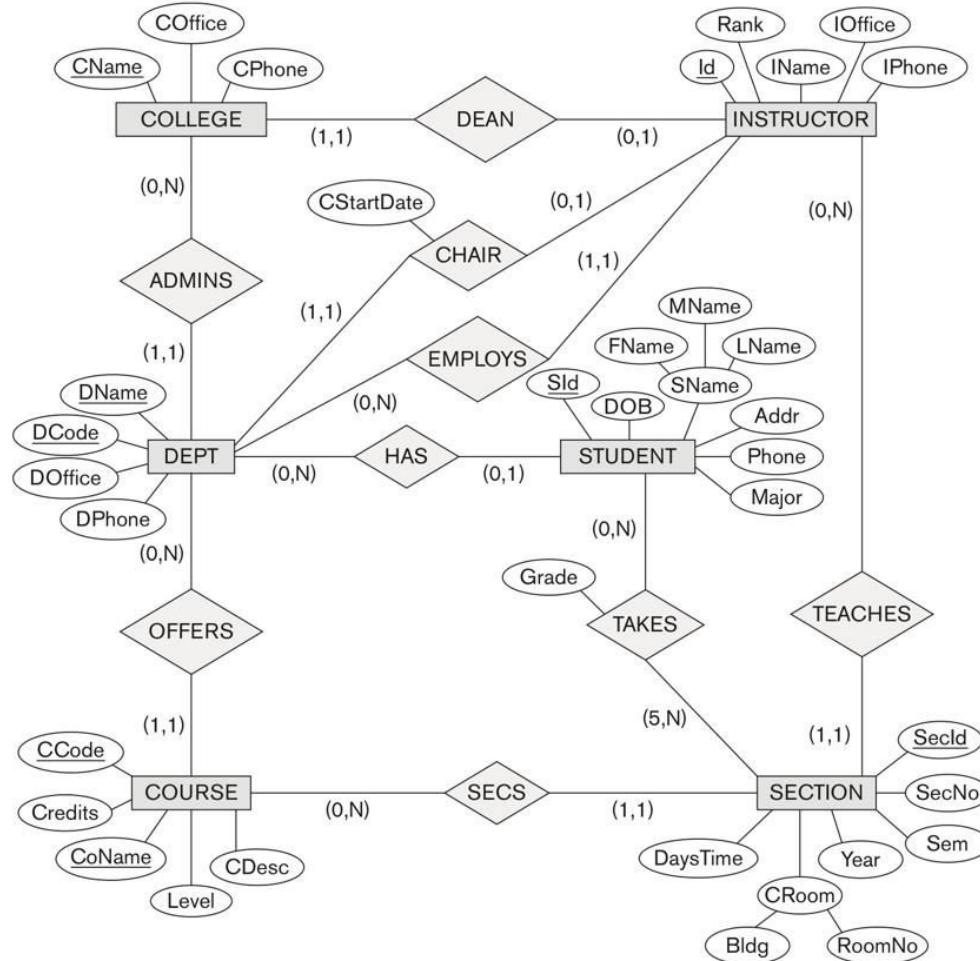
Displaying constraints on higher-degree relationships

- The (min, max) constraints can be displayed on the edges
 - however, they do not fully describe the constraints
- Displaying a 1, M, or N indicates additional constraints
 - An M or N indicates no constraint
 - A 1 indicates that an entity can participate in at most one relationship instance *that has a particular combination of the other participating entities*
- In general, both (min, max) and 1, M, or N are needed to describe fully the constraints
- Overall, the constraint specification is difficult and possibly ambiguous when we consider relationships of a degree higher than two.

Another Example: A UNIVERSITY Database

- To keep track of the enrollments in classes and student grades, another database is to be designed.
- It keeps track of the COLLEGES, DEPARTMENTS within each college, the COURSEs offered by departments, and SECTIONS of courses, INSTRUCTORs who teach the sections etc.
- These entity types and the relationships among these entity types are shown on the next slide in Figure 3.20.

UNIVERSITY database conceptual schema



Chapter Summary

- ER Model Concepts: Entities, attributes, relationships
- Constraints in the ER model
- Using ER in step-by-step mode conceptual schema design for the COMPANY database
- ER Diagrams - Notation
- Alternative Notations – UML class diagrams, others
- Binary Relationship types and those of higher degree.

Data Modeling Tools (Additional Material)

- 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
- NEGATIVES:
 - Most tools lack a proper distinct notation for relationships with relationship attributes
 - Mostly represent a relational design in a diagrammatic form rather than a conceptual ER-based design

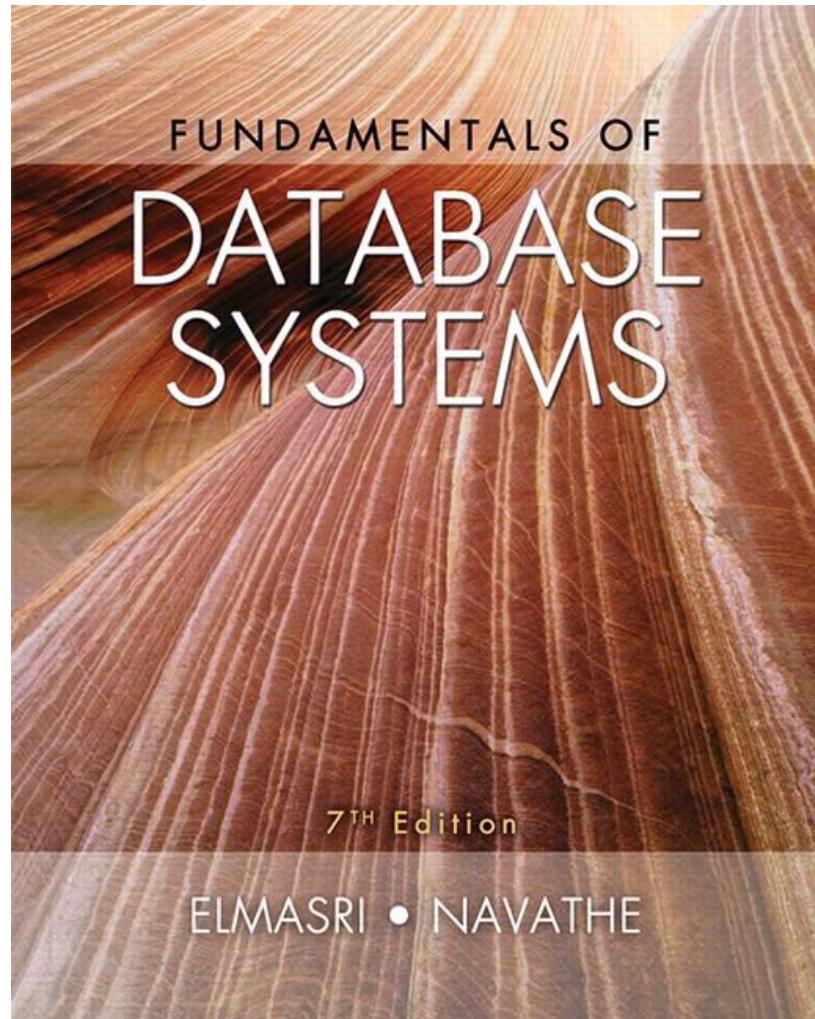
Some of the Automated Database Design Tools

(Note: Not all may be on the market now)

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

Extended Entity-Relationship (EER) Model (in the next chapter)

- The entity relationship model in its original form did not support the specialization and generalization abstractions
- Next chapter illustrates how the ER model can be extended with
 - Type-subtype and set-subset relationships
 - Specialization/Generalization Hierarchies
 - Notation to display them in EER diagrams



CHAPTER 4

Enhanced Entity-Relationship (EER) Modeling

Chapter Outline

- EER stands for Enhanced ER or Extended ER
- EER Model Concepts
 - Includes all modeling concepts of basic ER
 - Additional concepts:
 - subclasses/superclasses
 - specialization/generalization
 - categories (UNION types)
 - attribute and relationship inheritance
 - Constraints on Specialization/Generalization
- The additional EER concepts are used to model applications more completely and more accurately
 - EER includes some object-oriented concepts, such as inheritance
- Knowledge Representation and Ontology Concepts

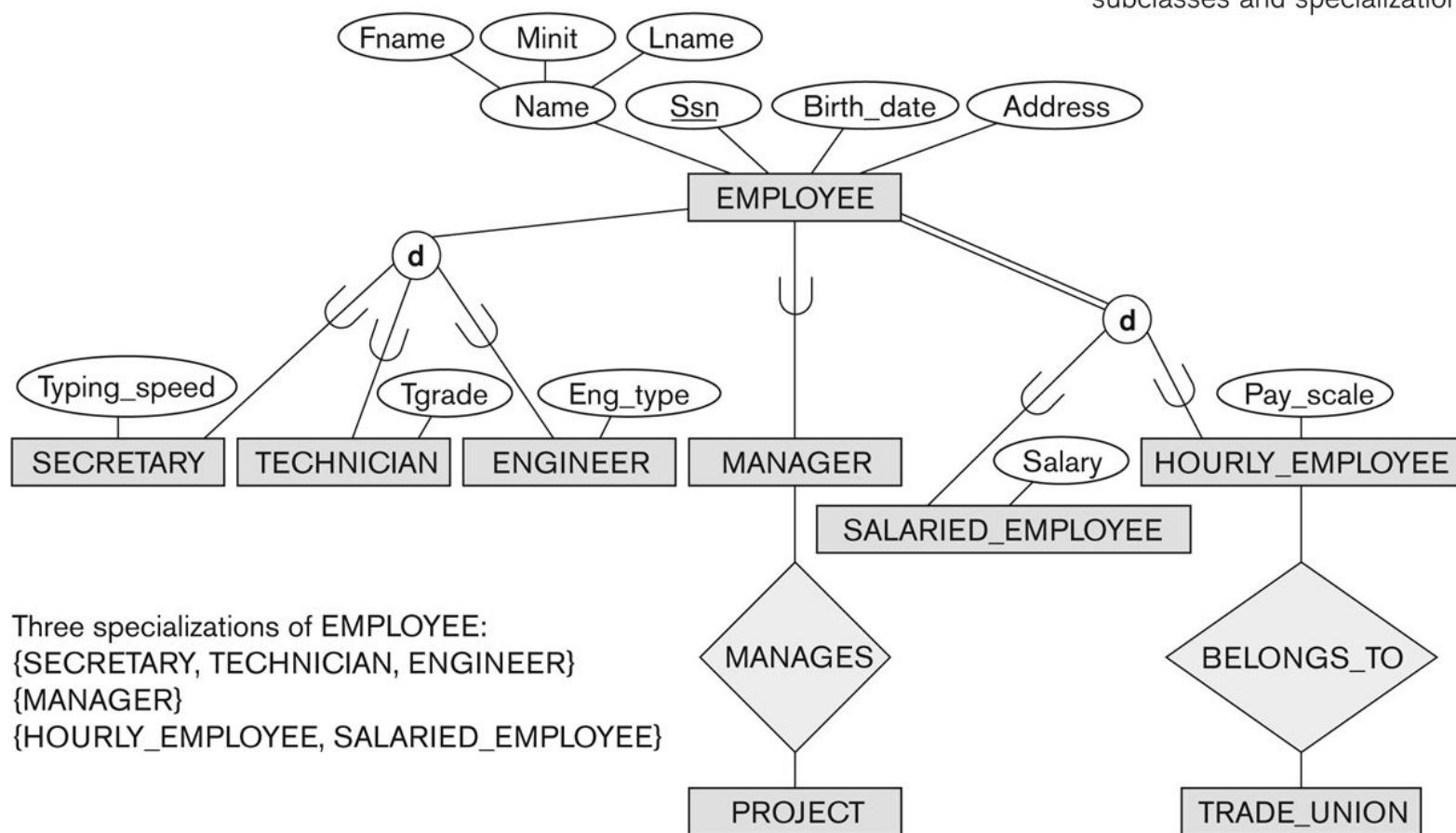
Subclasses and Superclasses (1)

- An entity type may have additional meaningful subgroupings of its entities
 - Example: EMPLOYEE may be further grouped into:
 - SECRETARY, ENGINEER, TECHNICIAN, ...
 - Based on the EMPLOYEE's Job
 - MANAGER
 - EMPLOYEES who are managers (the role they play)
 - SALARIED_EMPLOYEE, HOURLY_EMPLOYEE
 - Based on the EMPLOYEE's method of pay
- EER diagrams extend ER diagrams to represent these additional subgroupings, called *subclasses* or *subtypes*

Subclasses and Superclasses

Figure 4.1

EER diagram notation to represent subclasses and specialization.



Subclasses and Superclasses (2)

- Each of these subgroupings is a subset of EMPLOYEE entities
- Each is called a subclass of EMPLOYEE
- EMPLOYEE is the superclass for each of these subclasses
- These are called superclass/subclass relationships:
 - EMPLOYEE/SECRETARY
 - EMPLOYEE/TECHNICIAN
 - EMPLOYEE/MANAGER
 - ...

Subclasses and Superclasses (3)

- These are also called IS-A relationships
 - SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE,
- Note: An entity that is member of a subclass represents the same real-world entity as some member of the superclass:
 - The subclass member is the same entity in a *distinct specific role*
 - An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass
 - A member of the superclass can be optionally included as a member of any number of its subclasses

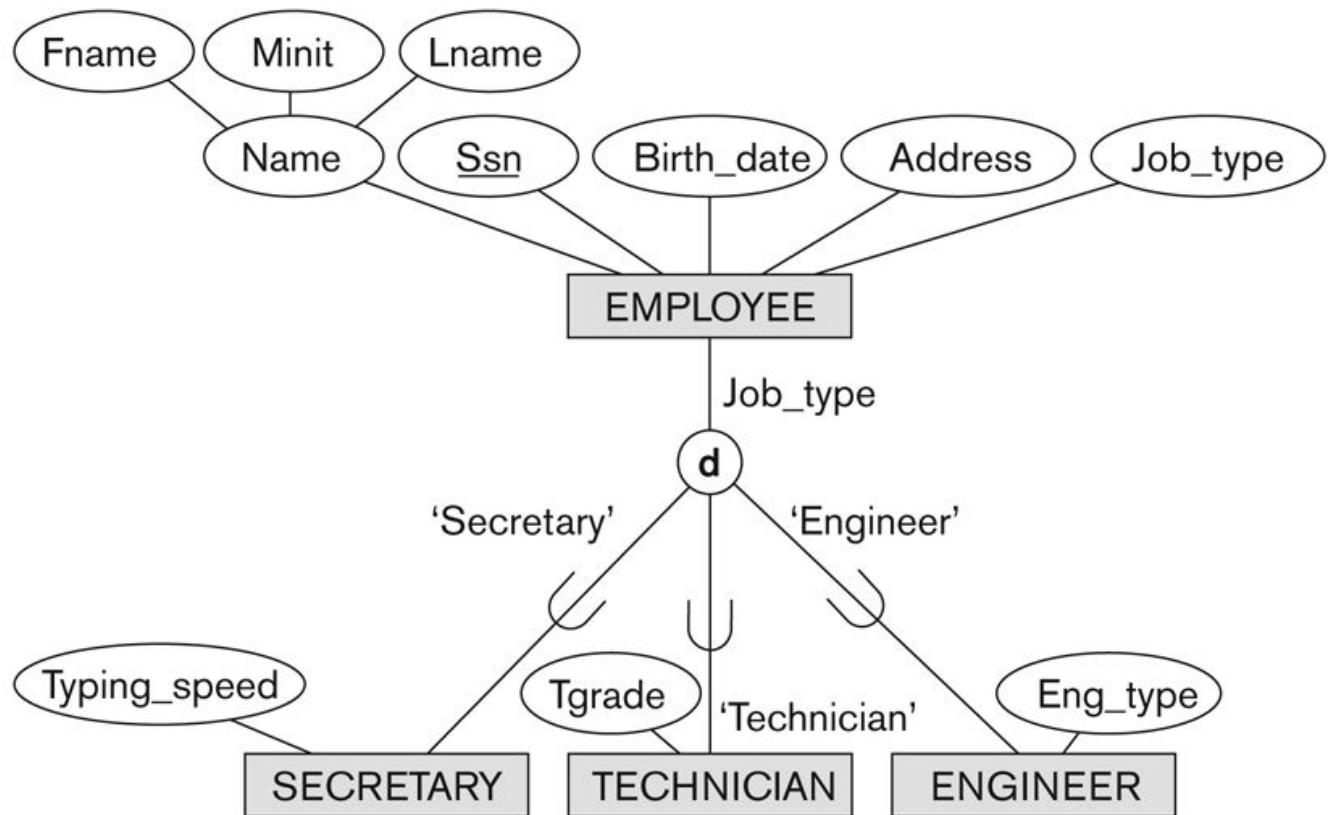
Subclasses and Superclasses (4)

- Examples:
 - A salaried employee who is also an engineer belongs to the two subclasses:
 - ENGINEER, and
 - SALARIED_EMPLOYEE
 - A salaried employee who is also an engineering manager belongs to the three subclasses:
 - MANAGER,
 - ENGINEER, and
 - SALARIED_EMPLOYEE
- It is not necessary that every entity in a superclass be a member of some subclass

Representing Specialization in EER Diagrams

Figure 4.4

EER diagram notation for an attribute-defined specialization on Job_type.



Attribute Inheritance in Superclass / Subclass Relationships

- An entity that is member of a subclass *inherits*
 - All attributes of the entity as a member of the superclass
 - All relationships of the entity as a member of the superclass
- Example:
 - In the previous slide, SECRETARY (as well as TECHNICIAN and ENGINEER) inherit the attributes Name, SSN, ..., from EMPLOYEE
 - Every SECRETARY entity will have values for the inherited attributes

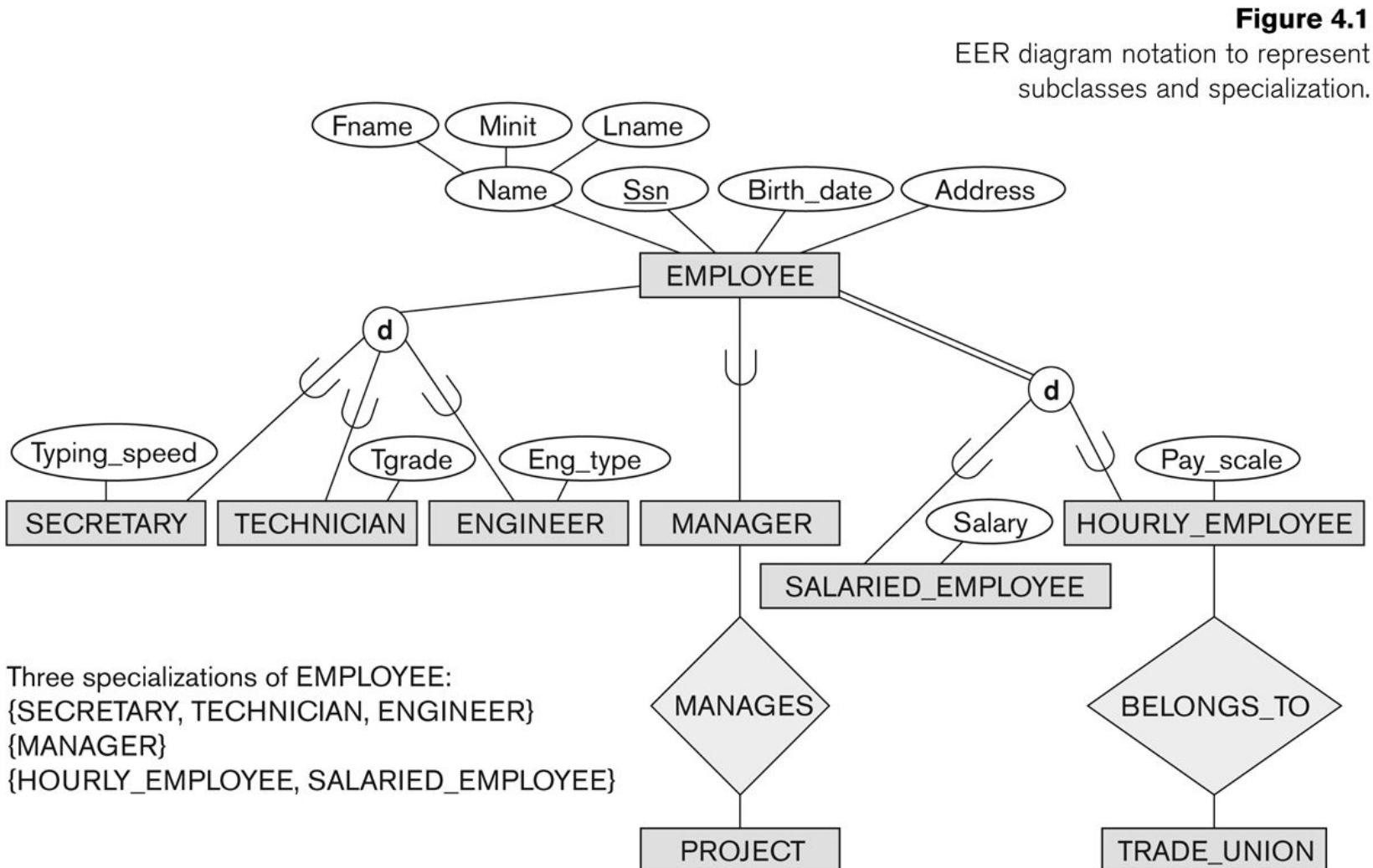
Specialization (1)

- Specialization is the process of defining a set of subclasses of a superclass
- The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass
 - Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon *job type*.
 - Example: MANAGER is a specialization of EMPLOYEE based on the role the employee plays
 - May have several specializations of the same superclass

Specialization (2)

- Example: Another specialization of EMPLOYEE based on *method of pay* is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.
 - Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams
 - Attributes of a subclass are called *specific* or *local* attributes.
 - For example, the attribute TypingSpeed of SECRETARY
 - The subclass can also participate in specific relationship types.
 - For example, a relationship BELONGS_TO of HOURLY_EMPLOYEE

Specialization (3)

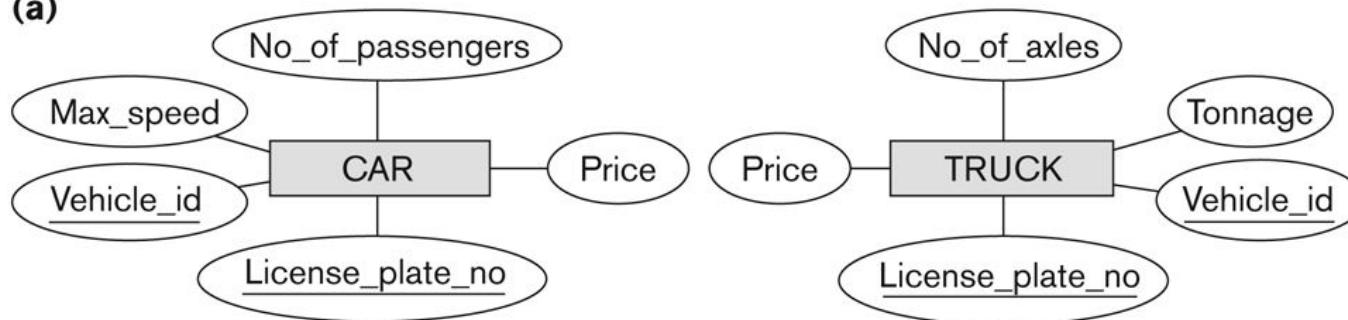


Generalization

- Generalization is the reverse of the specialization process
- Several classes with common features are generalized into a superclass:
 - original classes become its subclasses
- Example: CAR, TRUCK generalized into VEHICLE;
 - both CAR, TRUCK become subclasses of the superclass VEHICLE.
 - We can view {CAR, TRUCK} as a specialization of VEHICLE
 - Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK

Generalization (2)

(a)



(b)

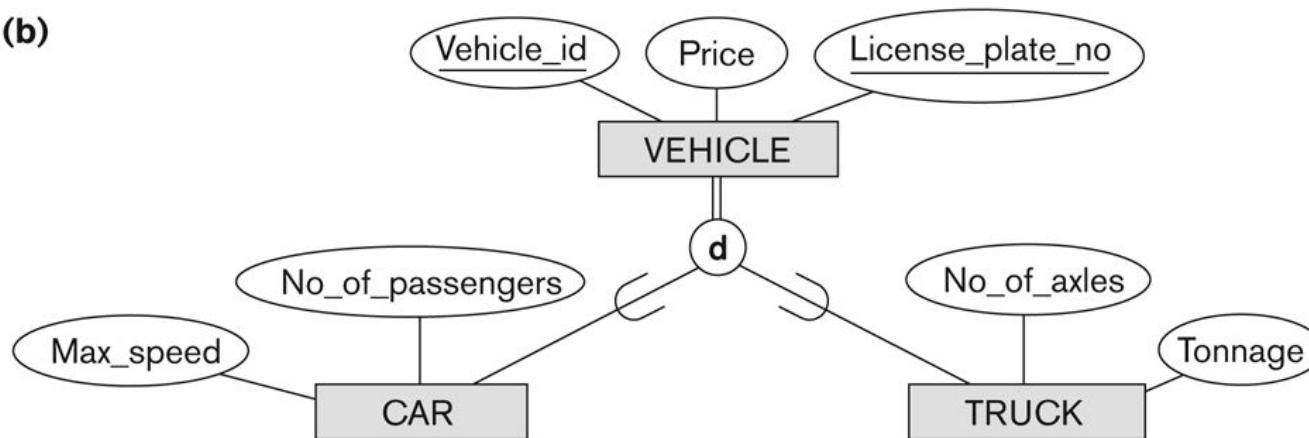


Figure 4.3

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

Generalization and Specialization (1)

- Diagrammatic notations are sometimes used to distinguish between generalization and specialization
 - Arrow pointing to the generalized superclass represents a generalization
 - Arrows pointing to the specialized subclasses represent a specialization
 - We *do not use* this notation because it is often subjective as to which process is more appropriate for a particular situation
 - We advocate not drawing any arrows

Generalization and Specialization (2)

- Data Modeling with Specialization and Generalization
 - A superclass or subclass represents a collection (or set or grouping) of entities
 - It also represents a particular *type of entity*
 - Shown in rectangles in EER diagrams (as are entity types)
 - We can call all entity types (and their corresponding collections) **classes**, whether they are entity types, superclasses, or subclasses

Types of Specialization

- Predicate-defined (or condition-defined) : based on some predicate. E.g., based on value of an attribute, say, Job-type, or Age.
- Attribute-defined: shows the name of the attribute next to the line drawn from the superclass toward the subclasses (see Fig. 4.1)
- User-defined: membership is defined by the user on an entity by entity basis

Constraints on Specialization and Generalization (1)

- If we can determine exactly those entities that will become members of each subclass by a condition, the subclasses are called predicate-defined (or condition-defined) subclasses
 - Condition is a constraint that determines subclass members
 - Display a predicate-defined subclass by writing the predicate condition next to the line attaching the subclass to its superclass

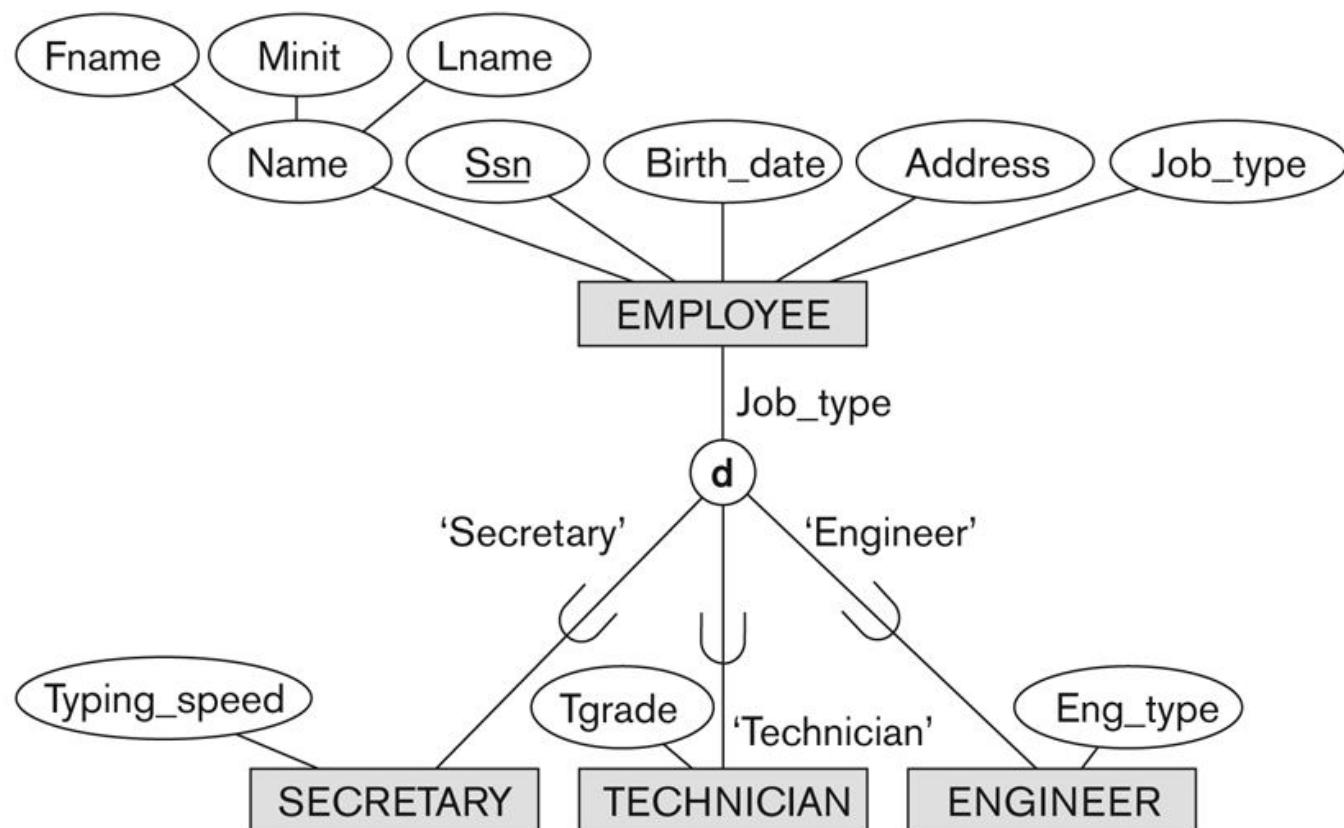
Constraints on Specialization and Generalization (2)

- If all subclasses in a specialization have membership condition on same attribute of the superclass, specialization is called an attribute-defined specialization
 - Attribute is called the defining attribute of the specialization
 - Example: JobType is the defining attribute of the specialization {SECRETARY, TECHNICIAN, ENGINEER} of EMPLOYEE
- If no condition determines membership, the subclass is called user-defined
 - Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass
 - Membership in the subclass is specified individually for each entity in the superclass by the user

Displaying an attribute-defined specialization in EER diagrams

Figure 4.4

EER diagram notation for an attribute-defined specialization on Job_type.



Constraints on Specialization and Generalization (3)

- Two basic constraints can apply to a specialization/generalization:
 - Disjointness Constraint:
 - Completeness Constraint:

Constraints on Specialization and Generalization (4)

- Disjointness Constraint:
 - Specifies that the subclasses of the specialization must be *disjoint*:
 - an entity can be a member of at most one of the subclasses of the specialization
 - Specified by d in EER diagram
 - If not disjoint, specialization is *overlapping*:
 - that is the same entity may be a member of more than one subclass of the specialization
 - Specified by o in EER diagram

Constraints on Specialization and Generalization (5)

- Completeness (Exhaustiveness) Constraint:
 - *Total* specifies that every entity in the superclass must be a member of some subclass in the specialization/generalization
 - Shown in EER diagrams by a **double line**
 - *Partial* allows an entity not to belong to any of the subclasses
 - Shown in EER diagrams by a single line

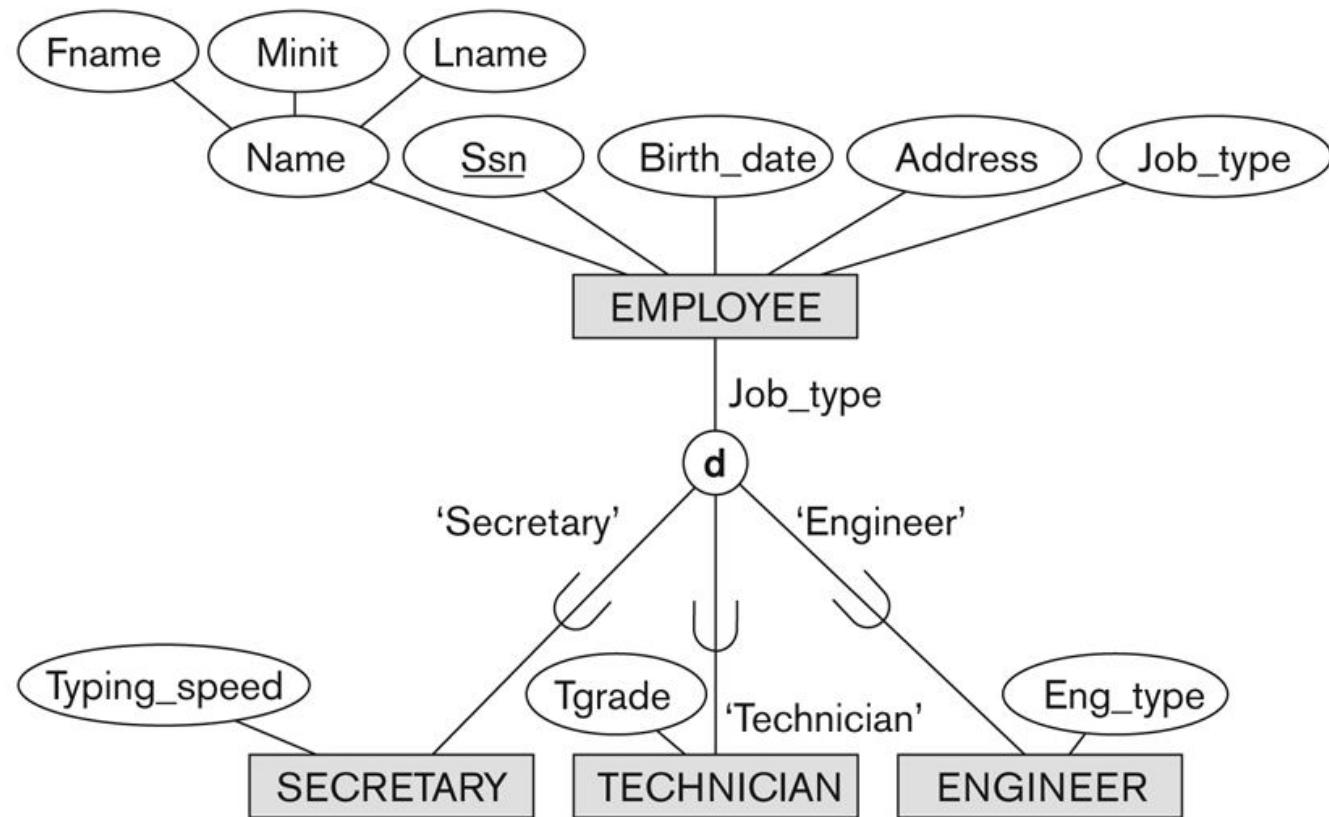
Constraints on Specialization and Generalization (6)

- Hence, we have four types of specialization/generalization:
 - Disjoint, total
 - Disjoint, partial
 - Overlapping, total
 - Overlapping, partial
- Note: Generalization usually is total because the superclass is derived from the subclasses.

Example of disjoint partial Specialization

Figure 4.4

EER diagram notation for an attribute-defined specialization on Job_type.



Example of overlapping total Specialization

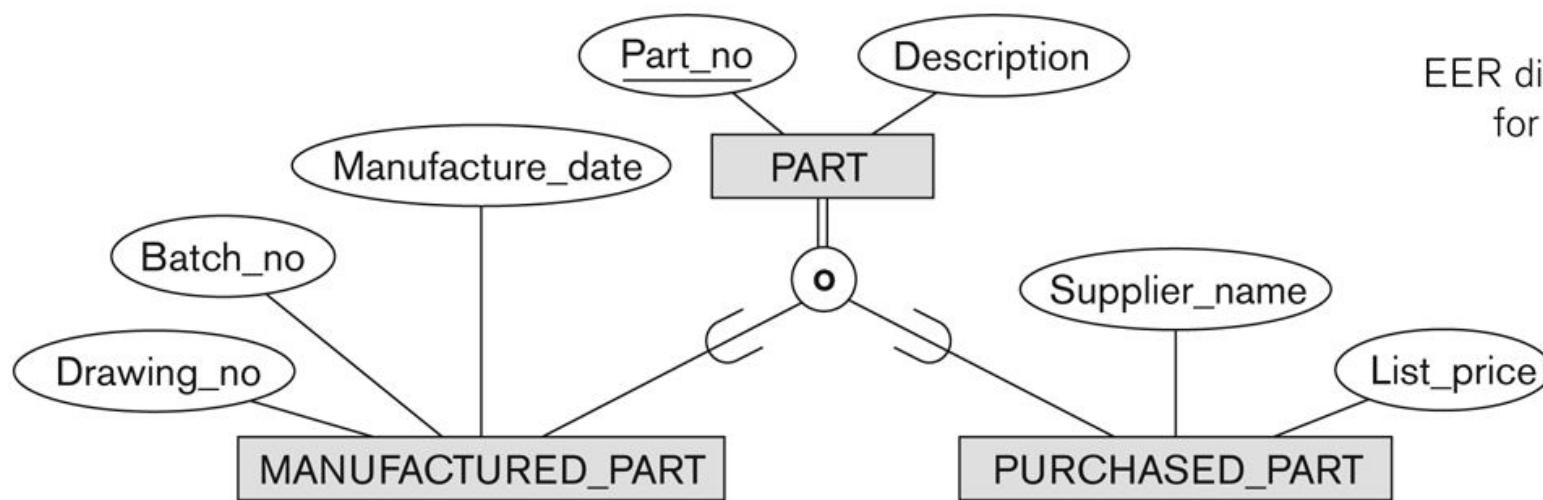


Figure 4.5
EER diagram notation
for an overlapping
(nondisjoint)
specialization.

Specialization/Generalization Hierarchies, Lattices & Shared Subclasses (1)

- A subclass may itself have further subclasses specified on it
 - forms a hierarchy or a lattice
- **Hierarchy** has a constraint that every subclass has only one superclass (called **single inheritance**); this is basically a **tree structure**
- In a **lattice**, a subclass can be subclass of more than one superclass (called **multiple inheritance**)

Shared Subclass “Engineering_Manager”

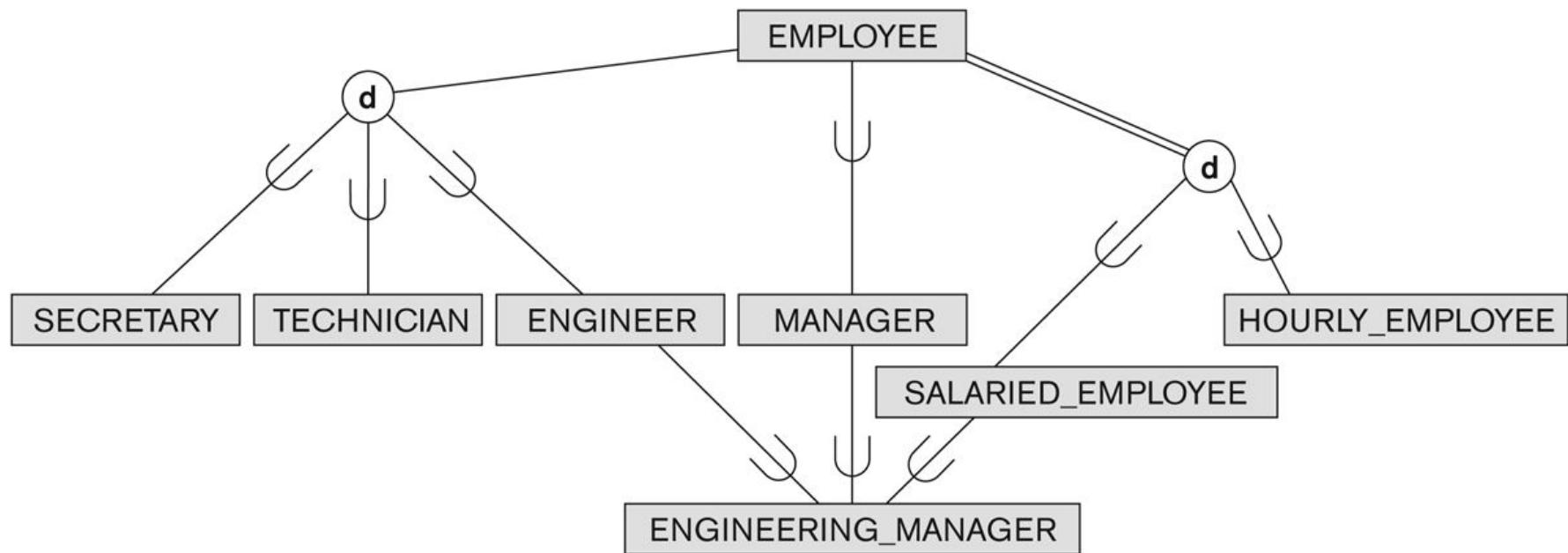


Figure 4.6
A specialization lattice with shared subclass ENGINEERING_MANAGER.

Specialization/Generalization Hierarchies, Lattices & Shared Subclasses (2)

- In a lattice or hierarchy, a subclass inherits attributes not only of its direct superclass, but also of all its predecessor superclasses
- A subclass with more than one superclass is called a shared subclass (multiple inheritance)
- Can have:
 - *specialization* hierarchies or lattices, or
 - *generalization* hierarchies or lattices,
 - depending on how they were *derived*
- We just use *specialization* (to stand for the end result of either specialization or generalization)

Specialization/Generalization Hierarchies, Lattices & Shared Subclasses (3)

- In *specialization*, start with an entity type and then define subclasses of the entity type by successive specialization
 - called a *top down* conceptual refinement process
- In *generalization*, start with many entity types and generalize those that have common properties
 - Called a *bottom up* conceptual synthesis process
- In practice, a *combination of both processes* is usually employed

Specialization / Generalization Lattice

Example (UNIVERSITY)

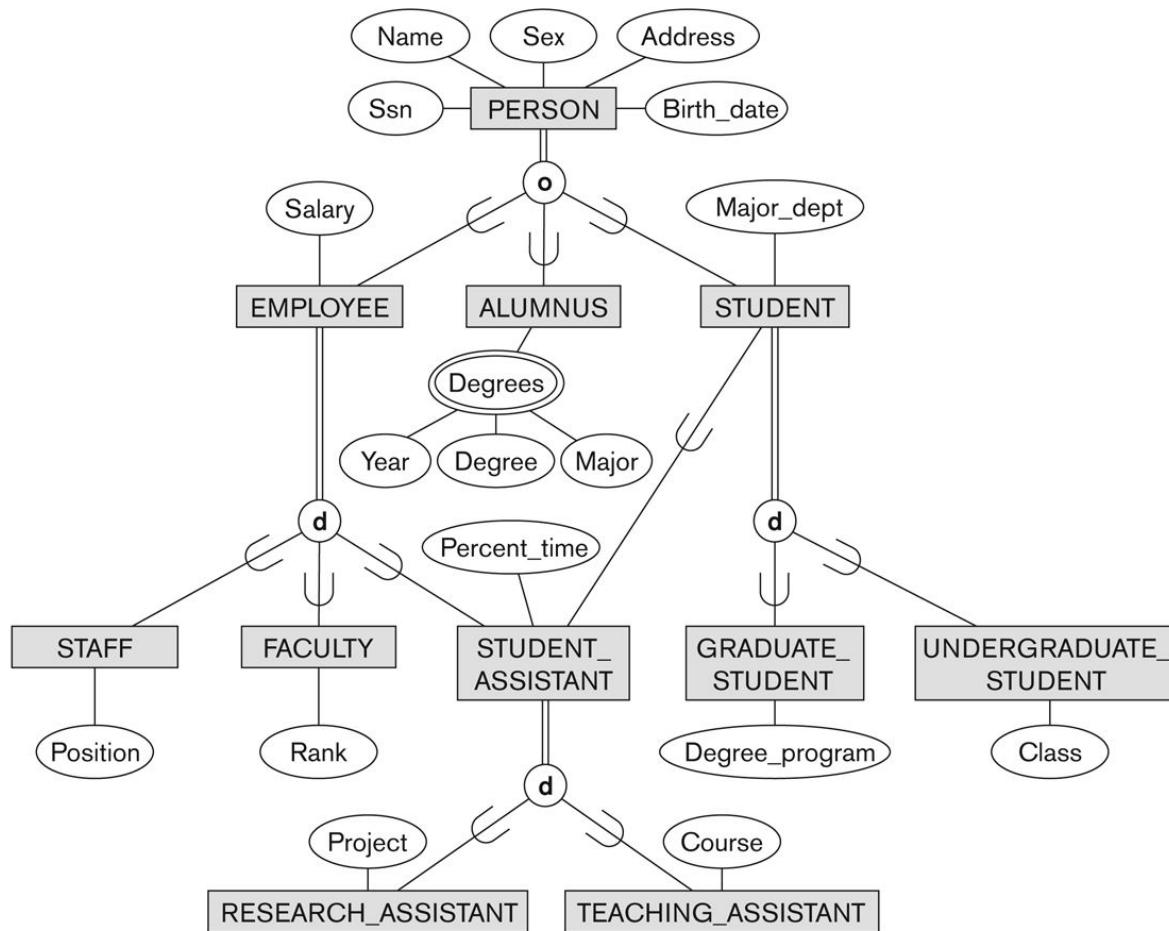


Figure 4.7

A specialization lattice with multiple inheritance for a UNIVERSITY database.

Categories (UNION TYPES) (1)

- All of the *superclass/subclass relationships* we have seen thus far have a single superclass
- A shared subclass is a subclass in:
 - *more than one* distinct superclass/subclass relationships
 - each relationships has a *single* superclass
 - shared subclass leads to multiple inheritance
- In some cases, we need to model a *single superclass/subclass relationship with more than one superclass*
- Superclasses can represent different entity types
- Such a subclass is called a category or UNION TYPE

Categories (UNION TYPES) (2)

- Example: In a database for vehicle registration, a vehicle owner can be a PERSON, a BANK (holding a lien on a vehicle) or a COMPANY.
 - A category (UNION type) called OWNER is created to represent a subset of the *union* of the three superclasses COMPANY, BANK, and PERSON
 - A category member must exist in ***at least one (typically just one)*** of its superclasses
- Difference from *shared subclass*, which is a:
 - subset of the *intersection* of its superclasses
 - shared subclass member must exist in ***all*** of its superclasses

Two categories (UNION types): OWNER, REGISTERED_VEHICLE

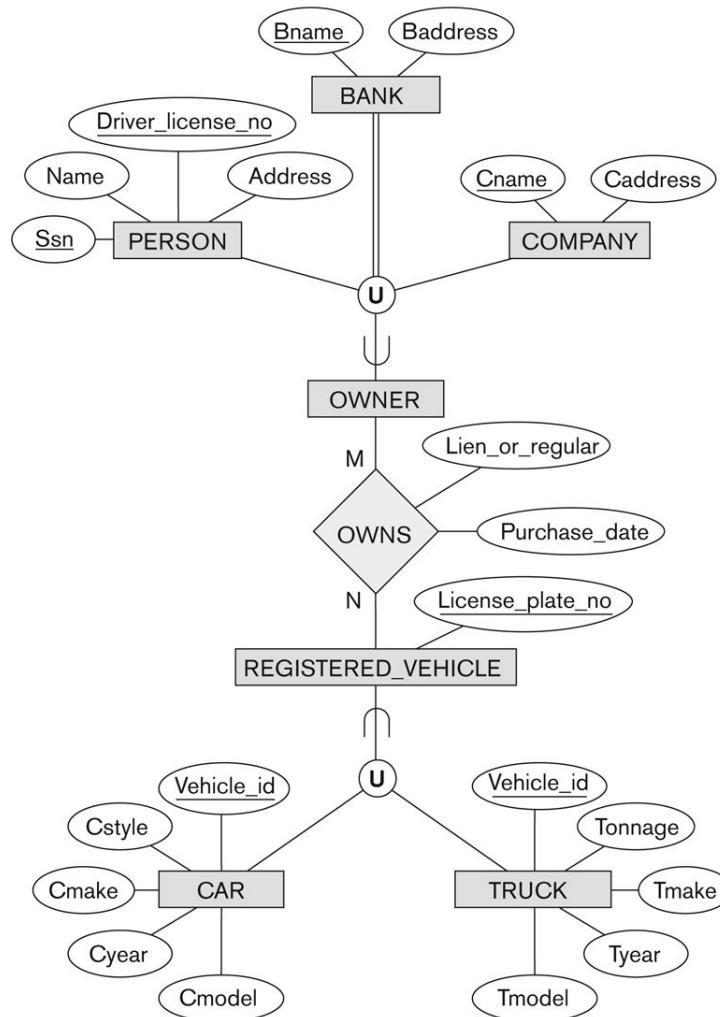


Figure 4.8
Two categories (union types): OWNER and REGISTERED_VEHICLE.

Formal Definitions of EER Model (1)

- Class C:
 - A type of entity with a corresponding set of entities:
 - could be entity type, subclass, superclass, or category
- Note: The definition of *relationship type* in ER/EER should have 'entity type' replaced with 'class' to allow relationships among classes in general
- Subclass S is a class whose:
 - Type inherits all the attributes and relationship of a class C
 - Set of entities must always be a subset of the set of entities of the other class C
 - $S \subseteq C$
 - C is called the superclass of S
 - A superclass/subclass relationship exists between S and C

Formal Definitions of EER Model (2)

- Specialization Z: $Z = \{S_1, S_2, \dots, S_n\}$ is a set of subclasses with same superclass G; hence, G/S_i is a superclass relationship for $i = 1, \dots, n$.
 - G is called a generalization of the subclasses $\{S_1, S_2, \dots, S_n\}$
 - Z is total if we always have:
 - $S_1 \cup S_2 \cup \dots \cup S_n = G$;
 - Otherwise, Z is partial.
 - Z is disjoint if we always have:
 - $S_i \cap S_j$ empty-set for $i \neq j$;
 - Otherwise, Z is overlapping.

Formal Definitions of EER Model (3)

- Subclass S of C is predicate defined if predicate (condition) p on attributes of C is used to specify membership in S;
 - that is, $S = C[p]$, where $C[p]$ is the set of entities in C that satisfy condition p
- A subclass not defined by a predicate is called user-defined
- Attribute-defined specialization: if a predicate $A = ci$ (where A is an attribute of G and ci is a constant value from the domain of A) is used to specify membership in each subclass Si in Z
 - Note: If $ci \neq cj$ for $i \neq j$, and A is single-valued, then the attribute-defined specialization will be disjoint.

Formal Definitions of EER Model (4)

- Category or UNION type T
 - A class that is a subset of the *union* of n defining superclasses
 $D_1, D_2, \dots, D_n, n > 1:$
 - $T \subseteq (D_1 \cup D_2 \cup \dots \cup D_n)$
 - Can have a predicate p_i on the attributes of D_i to specify entities of D_i that are members of T .
 - If a predicate is specified on every D_i : $T = (D_1[p_1] \cup D_2[p_2] \cup \dots \cup D_n[p_n])$

Alternative diagrammatic notations

- ER/EER diagrams are a specific notation for displaying the concepts of the model diagrammatically
- DB design tools use many alternative notations for the same or similar concepts
- One popular alternative notation uses *UML class diagrams*
- see next slides for UML class diagrams and other alternative notations

UML Example for Displaying Specialization / Generalization

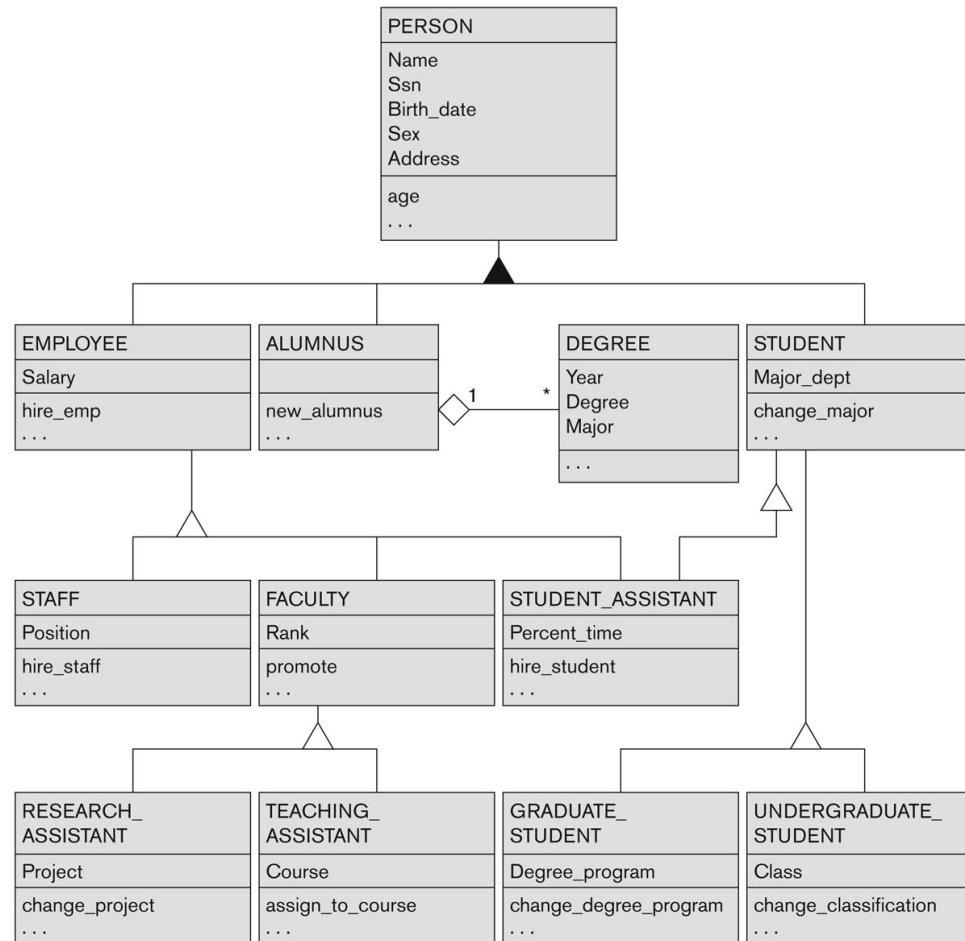


Figure 4.10

A UML class diagram corresponding to the EER diagram in Figure 4.7, illustrating UML notation for specialization/generalization.

Alternative Diagrammatic Notations

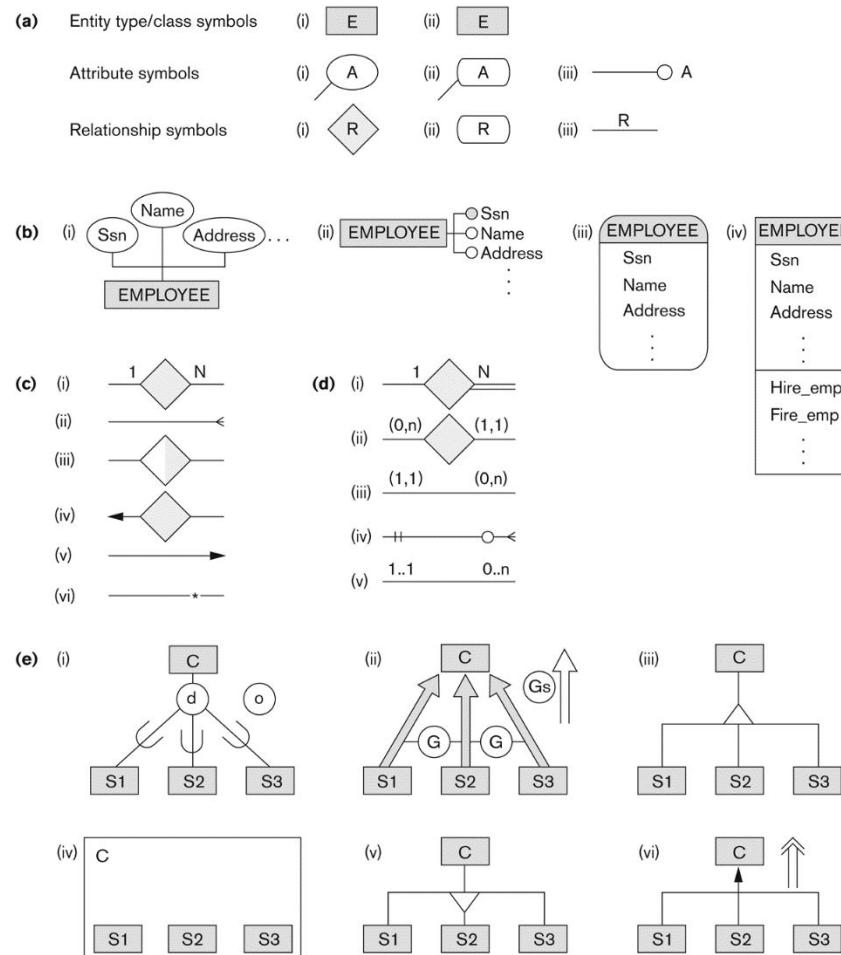


Figure A.1

Alternative notations. (a) Symbols for entity type/class, attribute, and relationship. (b) Displaying attributes. (c) Displaying cardinality ratios. (d) Various (min, max) notations. (e) Notations for displaying specialization/generalization.

Knowledge Representation (KR)-1

- Deals with modeling and representing a certain domain of knowledge.
- Typically done by using some formal model of representation and by creating an Ontology
- An ontology for a specific domain of interest describes a set of concepts and interrelationships among those concepts
- An Ontology serves as a “schema” which enables interpretation of the knowledge in a “knowledge-base”

Knowledge Representation (KR)-2

COMMON FEATURES between KR and Data Models:

- Both use similar set of abstractions – classification, aggregation, generalization, and identification.
- Both provide concepts, relationships, constraints, operations and languages to represent knowledge and model data

DIFFERENCES:

- KR has broader scope: tries to deal with missing and incomplete knowledge, default and common-sense knowledge etc.

Knowledge Representation (KR)-3

DIFFERENCES (continued):

- KR schemes typically include rules and reasoning mechanisms for inferencing
- Most KR techniques involve data and metadata. In data modeling, these are treated separately
- KR is used in conjunction with artificial intelligence systems to do decision support applications

For more details on spatial, temporal and multimedia data modeling, see Chapter 26. For details on use of Ontologies see Sections 27.4.3 and 27.7.4.

General Basis for Conceptual Modeling

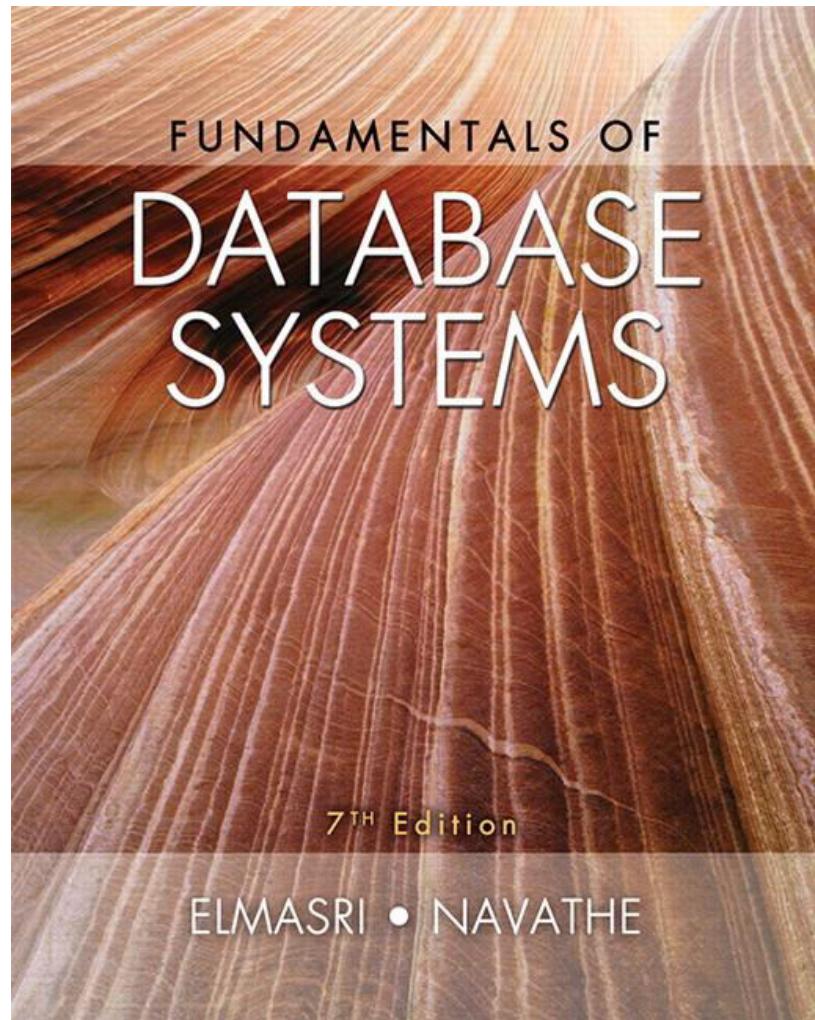
- TYPES OF DATA ABSTRACTIONS
 - CLASSIFICATION and INSTANTIATION
 - AGGREGATION and ASSOCIATION
(relationships)
 - GENERALIZATION and SPECIALIZATION
 - IDENTIFICATION
- CONSTRAINTS
 - CARDINALITY (Min and Max)
 - COVERAGE (Total vs. Partial, and Exclusive
(Disjoint) vs. Overlapping)

Ontologies

- Use conceptual modeling and other tools to develop “a specification of a conceptualization”
 - **Specification** refers to the language and vocabulary (data model concepts) used
 - **Conceptualization** refers to the description (schema) of the concepts of a particular field of knowledge and the relationships among these concepts
- Many medical, scientific, and engineering ontologies are being developed as a means of standardizing concepts and terminology

Summary

- Introduced the EER model concepts
 - Class/subclass relationships
 - Specialization and generalization
 - Inheritance
- Constraints on EER schemas
- These augment the basic ER model concepts introduced in Chapter 3
- EER diagrams and alternative notations were presented
- Knowledge Representation and Ontologies were introduced and compared with Data Modeling



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*
 - 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 *formal relational model* in this chapter
- In *practice*, there is a *standard model* based on SQL – this is described in Chapters 6 and 7 as a language
- Note: There are several important differences between the *formal* model and the *practical* model, as we shall see

Relational Model Concepts

- A Relation is a mathematical concept based on the ideas of sets
- The model was first proposed by Dr. E.F. Codd of IBM Research 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 Dr. Codd the coveted ACM Turing Award

Informal Definitions

- Informally, a **relation** looks like a **table** of values.
- A relation typically contains a **set of rows**.
- The data elements in each **row** represent certain facts that correspond to a real-world **entity** or **relationship**
 - In the formal model, rows are called **tuples**
- Each **column** has a column header that gives an indication of the meaning of the data items in that column
 - In the formal model, the column header is called an **attribute name** (or just **attribute**)

Example of a Relation

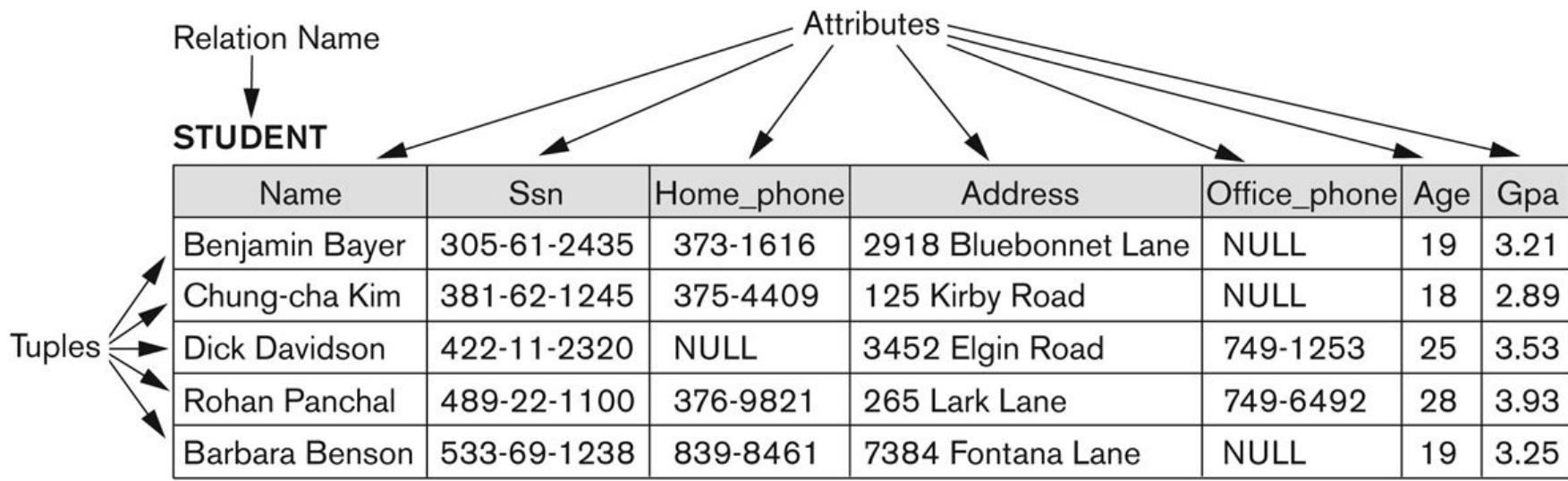


Figure 5.1
The attributes and tuples of a relation STUDENT.

Informal Definitions

■ Key of a Relation:

- Each row has a value of a data item (or set of items) that uniquely identifies that row in the table
 - Called the *key*
- In the STUDENT table, SSN is the key
- Sometimes row-ids or sequential numbers are assigned as keys to identify the rows in a table
 - Called *artificial key* or *surrogate key*

Formal Definitions - Schema

- The **Schema** (or description) of a Relation:
 - Denoted by $R(A_1, A_2, \dots, A_n)$
 - R is the **name** of the relation
 - The **attributes** of the relation are A_1, A_2, \dots, A_n
- Example:

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

 - **CUSTOMER** is the relation name
 - Defined over the four attributes: Cust-id, Cust-name, Address, Phone#
- Each attribute has a **domain** or a set of valid values.
 - For example, the domain of Cust-id is 6 digit numbers.

Formal Definitions - Tuple

- A **tuple** is an ordered set of values (enclosed in angled brackets '< ... >')
- Each value is derived from an appropriate *domain*.
- A row in the CUSTOMER relation is a 4-tuple and would consist of four values, for example:
 - <632895, "John Smith", "101 Main St. Atlanta, GA 30332", "(404) 894-2000">
 - This is called a 4-tuple as it has 4 values
 - A tuple (row) in the CUSTOMER relation.
- A relation is a **set** of such tuples (rows)

Formal Definitions - Domain

- A **domain** has a logical definition:
 - Example: “USA_phone_numbers” are the set of 10 digit phone numbers valid in the U.S.
- A domain also has 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.
 - Dates have various formats such as year, month, date formatted as yyyy-mm-dd, or as dd mm,yyyy etc.
- The attribute name designates the role played by a domain in a relation:
 - Used to interpret the meaning of the data elements corresponding to that attribute
 - Example: The domain Date may be used to define two attributes named “Invoice-date” and “Payment-date” with different meanings

Formal Definitions - State

- The **relation state** is a subset of the Cartesian product of the domains of its attributes
 - each domain contains the set of all possible values the attribute can take.
- Example: attribute Cust-name is defined over the domain of character strings of maximum length 25
 - $\text{dom}(\text{Cust-name})$ is $\text{varchar}(25)$
- The role these strings play in the CUSTOMER relation is that of the *name of a customer*.

Formal Definitions - Summary

- Formally,
 - Given $R(A_1, A_2, \dots, A_n)$
 - $r(R) \subset \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$
- $R(A_1, A_2, \dots, A_n)$ is the **schema** of the relation
- R is the **name** of the relation
- A_1, A_2, \dots, A_n are the **attributes** of the relation
- $r(R)$: a specific **state** (or "value" or "population") of relation R – this is a *set of tuples* (rows)
 - $r(R) = \{t_1, t_2, \dots, t_n\}$ where each t_i is an n -tuple
 - $t_i = \langle v_1, v_2, \dots, v_n \rangle$ where each v_j element-of $\text{dom}(A_j)$

Formal Definitions - Example

- Let $R(A_1, A_2)$ be a relation schema:
 - Let $\text{dom}(A_1) = \{0,1\}$
 - Let $\text{dom}(A_2) = \{a,b,c\}$
- Then: $\text{dom}(A_1) \times \text{dom}(A_2)$ is all possible combinations:
 $\{\langle 0,a \rangle, \langle 0,b \rangle, \langle 0,c \rangle, \langle 1,a \rangle, \langle 1,b \rangle, \langle 1,c \rangle\}$
- The relation state $r(R) \subset \text{dom}(A_1) \times \text{dom}(A_2)$
- For example: $r(R)$ could be $\{\langle 0,a \rangle, \langle 0,b \rangle, \langle 1,c \rangle\}$
 - this is one possible state (or “population” or “extension”) r of the relation R , defined over A_1 and A_2 .
 - It has three 2-tuples: $\langle 0,a \rangle, \langle 0,b \rangle, \langle 1,c \rangle$

Definition Summary

<u>Informal Terms</u>	<u>Formal Terms</u>
Table	Relation
Column Header	Attribute
All possible Column Values	Domain
Row	Tuple
Table Definition	Schema of a Relation
Populated Table	State of the Relation

Example – A relation STUDENT

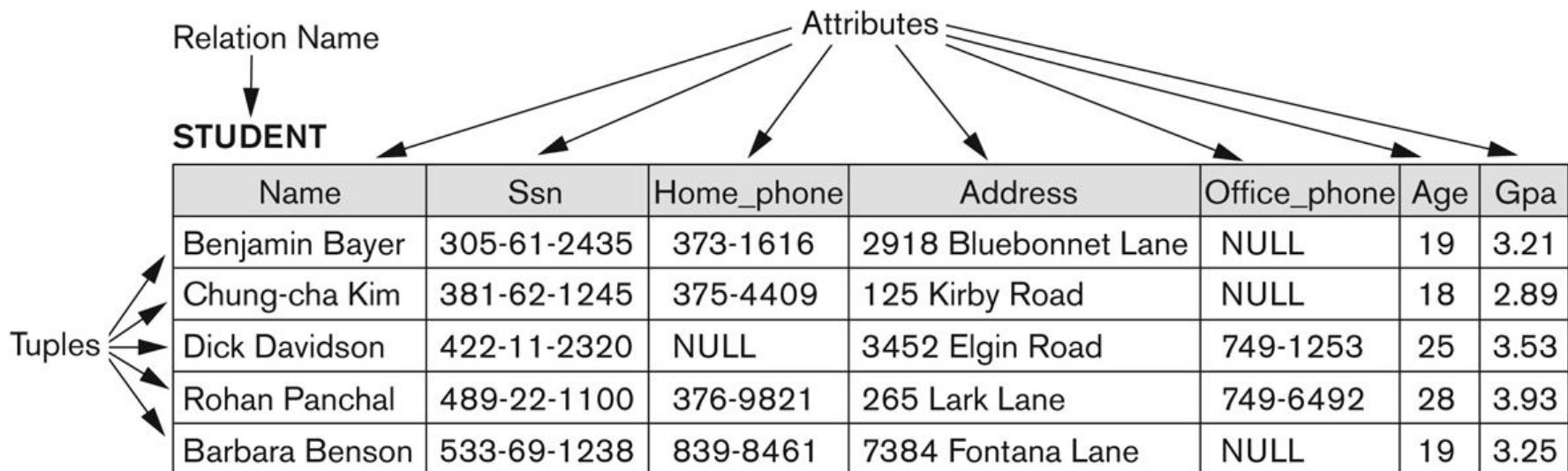


Figure 5.1

The attributes and tuples of a relation STUDENT.

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 $t = \langle v_1, v_2, \dots, v_n \rangle$ to be ordered .
 - (However, a more general alternative definition of relation does not require this ordering. It includes both the name and the value for each of the attributes).
 - Example: $t = \{ \langle \text{name}, \text{"John"} \rangle, \langle \text{SSN}, 123456789 \rangle \}$
 - This representation may be called as “self-describing”.

Same state as previous Figure (but with different order of tuples)

Figure 5.2

The relation STUDENT from Figure 5.1 with a different order of tuples.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	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
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Chung-cha Kim	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

Characteristics Of Relations

- Values in a tuple:
 - All values are considered atomic (indivisible).
 - Each value in a tuple must be from the domain of the attribute for that column
 - If tuple $t = \langle v_1, v_2, \dots, v_n \rangle$ is a tuple (row) in the relation state r of $R(A_1, A_2, \dots, A_n)$
 - Then each v_i must be a value from $\text{dom}(A_i)$
 - A special **null** value is used to represent values that are unknown or not available or inapplicable in certain tuples.

Characteristics Of Relations

- Notation:
 - We refer to **component values** of a tuple t by:
 - $t[A_i]$ or $t.A_i$
 - This is the value v_i 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 in t

CONSTRAINTS

Constraints determine which values are permissible and which are not in the database.

They are of three main types:

1. **Inherent or Implicit Constraints:** These are based on the data model itself. (E.g., relational model does not allow a list as a value for any attribute)
2. **Schema-based or Explicit Constraints:** They are expressed in the schema by using the facilities provided by the model. (E.g., max. cardinality ratio constraint in the ER model)
3. **Application based or semantic constraints:** These are beyond the expressive power of the model and must be specified and enforced by the application programs.

Relational Integrity Constraints

- Constraints are **conditions** that must hold on **all** valid relation states.
- There are three *main types* of (explicit schema-based) constraints that can be expressed in the relational model:
 - **Key** constraints
 - **Entity integrity** constraints
 - **Referential integrity** constraints
- Another schema-based constraint is the **domain** constraint
 - Every value in a tuple must be from the *domain of its attribute* (or it could be **null**, if allowed for that attribute)

Key Constraints

- **Superkey** of R:
 - Is a set of attributes SK of R with the following condition:
 - No two tuples in any valid relation state $r(R)$ will have the same value for SK
 - That is, for any distinct tuples t_1 and t_2 in $r(R)$, $t_1[SK] \neq t_2[SK]$
 - This condition must hold in *any valid state* $r(R)$
- **Key** of R:
 - A "minimal" superkey
 - That is, a key is a superkey K such that removal of any attribute from K results in a set of attributes that is not a superkey (does not possess the superkey uniqueness property)
- A Key is a Superkey but not vice versa

Key Constraints (continued)

- Example: Consider the CAR relation schema:
 - CAR(State, Reg#, SerialNo, Make, Model, Year)
 - CAR has two keys:
 - Key1 = {State, Reg#}
 - Key2 = {SerialNo}
 - Both are also superkeys of CAR
 - {SerialNo, Make} is a superkey but *not* a key.
- In general:
 - Any *key* is a *superkey* (but not vice versa)
 - Any set of attributes that *includes a key* is a *superkey*
 - A *minimal* superkey is also a key

Key Constraints (continued)

- If a relation has several **candidate keys**, one is chosen arbitrarily to be the **primary key**.
 - The primary key attributes are underlined.
- Example: Consider the CAR relation schema:
 - CAR(State, Reg#, SerialNo, Make, Model, Year)
 - We chose SerialNo as the primary key
- The primary key value is used to *uniquely identify* each tuple in a relation
 - Provides the tuple identity
- Also used to *reference* the tuple from another tuple
 - General rule: Choose as primary key the smallest of the candidate keys (in terms of size)
 - Not always applicable – choice is sometimes subjective

CAR table with two candidate keys – LicenseNumber chosen as Primary Key

CAR

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

Figure 5.4

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

Relational Database Schema

■ Relational Database Schema:

- A set S of relation schemas that belong to the same database.
- S is the name of the whole **database schema**
- $S = \{R_1, R_2, \dots, R_n\}$ and a set IC of integrity constraints.
- R_1, R_2, \dots, R_n are the names of the individual **relation schemas** within the database S
- Following slide shows a COMPANY database schema with 6 relation schemas

COMPANY Database Schema

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

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

PROJECT

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

WORKS_ON

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

DEPENDENT

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

Figure 5.5

Schema diagram for
the COMPANY
relational database
schema.

Relational Database State

- A **relational database state** DB of S is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC.
- A relational database *state* is sometimes called a relational database *snapshot* or *instance*.
- We will not use the term *instance* since it also applies to single tuples.
- A database state that does not meet the constraints is an invalid state

Populated database state

- Each *relation* will have many tuples in its current relation state
- The *relational database state* is a union of all the individual relation states
- Whenever the database is changed, a new state arises
- Basic operations for changing the database:
 - INSERT a new tuple in a relation
 - DELETE an existing tuple from a relation
 - MODIFY an attribute of an existing tuple
- Next slide (Fig. 5.6) shows an example state for the COMPANY database schema shown in Fig. 5.5.

Populated database state for COMPANY

Figure 5.6

One possible database state for the COMPANY relational database schema.

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

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

WORKS_ON

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

PROJECT

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

DEPENDENT

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

Entity Integrity

- **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 $r(R)$
 - If PK has several attributes, null is not allowed in any of these attributes
 - Note: Other attributes of R may be 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**.

Referential Integrity

- Tuples in the **referencing relation R1** have attributes FK (called **foreign key** attributes) that reference the primary key attributes PK of the **referenced relation R2**.
 - A tuple t1 in R1 is said to **reference** a tuple t2 in R2 if $t1[FK] = t2[PK]$.
- A referential integrity constraint can be displayed in a relational database schema as a directed arc from R1.FK to R2.

Referential Integrity (or foreign key) Constraint

- Statement of the constraint
 - The value in the foreign key column (or columns) FK of the the **referencing relation R1** can be **either:**
 - (1) a value of an existing primary key value of a corresponding primary key PK in the **referenced relation R2**, or
 - (2) a **null**.
- In case (2), the FK in R1 should **not** be a part of its own primary key.

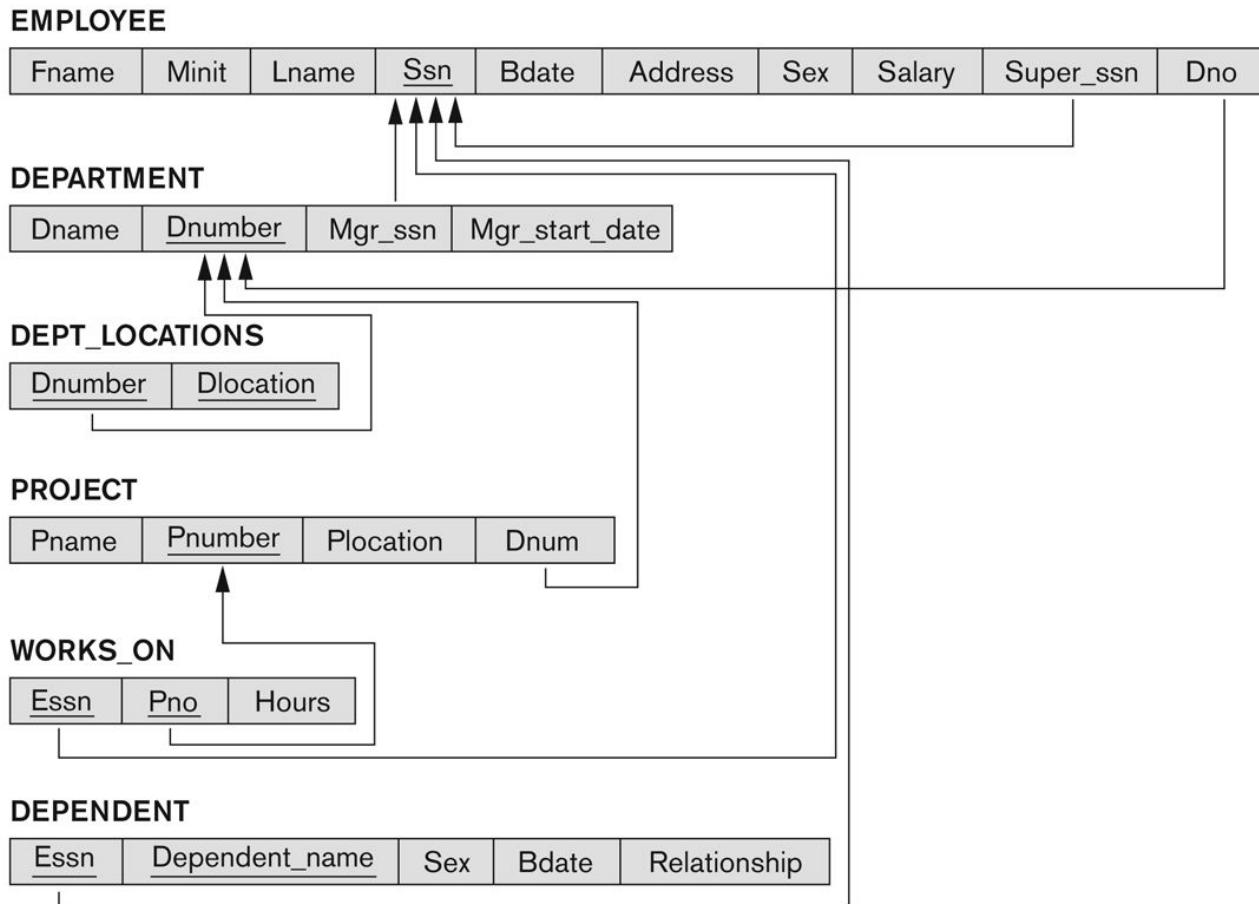
Displaying a relational database schema and its constraints

- Each relation schema can be displayed as a row of attribute names
- The name of the relation is written above the attribute names
- The primary key attribute (or attributes) will be underlined
- A foreign key (referential integrity) constraints is displayed as a directed arc (arrow) from the foreign key attributes to the referenced table
 - Can also point the the primary key of the referenced relation for clarity
- Next slide shows the **COMPANY relational schema diagram with referential integrity constraints**

Referential Integrity Constraints for COMPANY database

Figure 5.7

Referential integrity constraints displayed on the COMPANY relational database schema.



Other Types of Constraints

- Semantic Integrity Constraints:
 - based on application semantics and cannot be expressed by the model per se
 - Example: “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 **CREATE TRIGGER** and **CREATE ASSERTION** to express some of these semantic constraints
- Keys, Permissibility of Null values, Candidate Keys (Unique in SQL), Foreign Keys, Referential Integrity etc. are expressed by the **CREATE TABLE** statement in SQL.

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 (RESTRICT or 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

Possible violations for each operation

- INSERT may violate any of the constraints:
 - Domain constraint:
 - if one of the attribute values provided for the new tuple is not of the specified attribute domain
 - Key constraint:
 - if the value of a key attribute in the new tuple already exists in another tuple in the relation
 - Referential integrity:
 - if a foreign key value in the new tuple references a primary key value that does not exist in the referenced relation
 - Entity integrity:
 - if the primary key value is null in the new tuple

Possible violations for each operation

- DELETE may violate only referential integrity:
 - If the primary key value of the tuple being deleted is referenced from other tuples in the database
 - Can be remedied by several actions: RESTRICT, CASCADE, SET NULL (see Chapter 6 for more details)
 - RESTRICT option: reject the deletion
 - CASCADE option: propagate the new primary key value into the foreign keys of the referencing tuples
 - SET NULL option: set the foreign keys of the referencing tuples to NULL
 - One of the above options must be specified during database design for each foreign key constraint

Possible violations for each operation

- UPDATE may violate domain constraint and NOT NULL constraint on an attribute being modified
- Any of the other constraints may also be violated, depending on the attribute being updated:
 - Updating the primary key (PK):
 - Similar to a DELETE followed by an INSERT
 - Need to specify similar options to DELETE
 - Updating a foreign key (FK):
 - May violate referential integrity
 - Updating an ordinary attribute (neither PK nor FK):
 - Can only violate domain constraints

Summary

- Presented Relational Model Concepts
 - Definitions
 - Characteristics of relations
- Discussed Relational Model Constraints and Relational Database Schemas
 - Domain constraints
 - Key constraints
 - Entity integrity
 - Referential integrity
- Described the Relational Update Operations and Dealing with Constraint Violations

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)

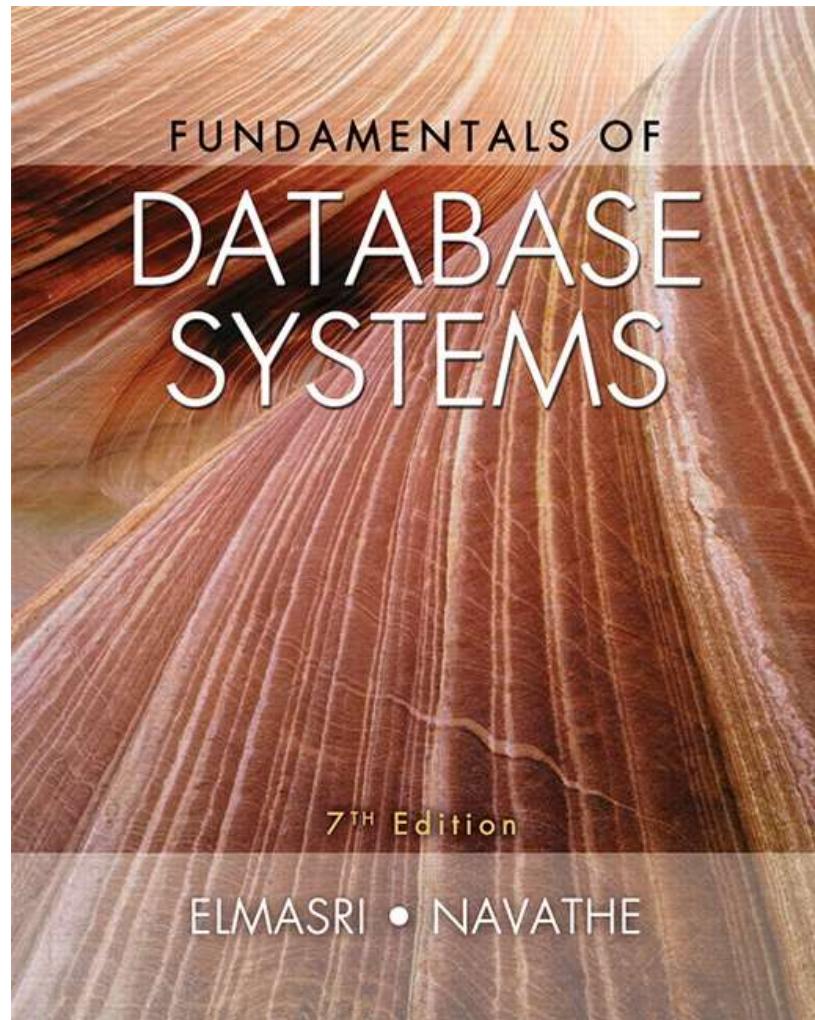
COURSE(Course#, Cname, Dept)

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.



CHAPTER 6

Basic SQL

Chapter 6 Outline

- SQL Data Definition and Data Types
- Specifying Constraints in SQL
- Basic Retrieval Queries in SQL
- INSERT, DELETE, and UPDATE Statements in SQL
- Additional Features of SQL

Basic SQL

- **SQL language**
 - Considered one of the major reasons for the commercial success of relational databases
- **SQL**
 - The origin of SQL is relational predicate calculus called tuple calculus (see Ch.8) which was proposed initially as the language SQUARE.
 - **SQL** Actually comes from the word “SEQUEL” which was the original term used in the paper: “SEQUEL TO SQUARE” by Chamberlin and Boyce. IBM could not copyright that term, so they abbreviated to SQL and copyrighted the term SQL.
 - Now popularly known as “Structured Query language”.
 - SQL is an informal or practical rendering of the relational data model with syntax

SQL Data Definition, Data Types, Standards

- Terminology:
 - **Table**, **row**, and **column** used for relational model terms relation, tuple, and attribute
- CREATE statement
 - Main SQL command for data definition
- The language has features for : Data definition, Data Manipulation, Transaction control (Transact-SQL, Ch. 20), Indexing (Ch.17), Security specification (Grant and Revoke- see Ch.30), Active databases (Ch.26), Multi-media (Ch.26), Distributed databases (Ch.23) etc.

SQL Standards

- SQL has gone through many standards: starting with SQL-86 or SQL 1.A. SQL-92 is referred to as SQL-2.
- Later standards (from SQL-1999) are divided into **core** specification and specialized **extensions**. The extensions are implemented for different applications – such as data mining, data warehousing, multimedia etc.
- SQL-2006 added XML features (Ch. 13); In 2008 they added Object-oriented features (Ch. 12).
- SQL-3 is the current standard which started with SQL-1999. It is not fully implemented in any RDBMS.

Schema and Catalog Concepts in SQL

- We cover the basic standard SQL syntax – there are variations in existing RDBMS systems
- **SQL schema**
 - Identified by a **schema name**
 - Includes an **authorization identifier** and **descriptors** for each element
- **Schema elements** include
 - Tables, constraints, views, domains, and other constructs
- Each statement in SQL ends with a **semicolon**

Schema and Catalog Concepts in SQL (cont'd.)

- CREATE SCHEMA statement
 - CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';
- Catalog
 - Named collection of schemas in an SQL environment
- SQL also has the concept of a cluster of catalogs.

The CREATE TABLE Command in SQL

- Specifying a new relation
 - Provide name of table
 - Specify attributes, their types and initial constraints
- Can optionally specify schema:
 - CREATE TABLE COMPANY.EMPLOYEE ...
or
 - CREATE TABLE EMPLOYEE ...

The CREATE TABLE Command in SQL (cont'd.)

- **Base tables (base relations)**
 - Relation and its tuples are actually created and stored as a file by the DBMS
- **Virtual relations (views)**
 - Created through the `CREATE VIEW` statement.
Do not correspond to any physical file.

COMPANY relational database schema (Fig. 5.7)

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

Dnumber	Dlocation
---------	-----------

PROJECT

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

WORKS_ON

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

DEPENDENT

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

One possible database state for the COMPANY relational database schema (Fig. 5.6)

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

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

One possible database state for the COMPANY relational database schema – continued (Fig. 5.6)

WORKS_ON

<u>Essn</u>	<u>Pno</u>	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

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

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

SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 5.7 (Fig. 6.1)

```
CREATE TABLE EMPLOYEE
( Fname          VARCHAR(15)      NOT NULL,
  Minit          CHAR,
  Lname          VARCHAR(15)      NOT NULL,
  Ssn            CHAR(9)         NOT NULL,
  Bdate          DATE,
  Address        VARCHAR(30),
  Sex            CHAR,
  Salary          DECIMAL(10,2),
  Super_ssn     CHAR(9),
  Dno            INT             NOT NULL,
  PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
( Dname          VARCHAR(15)      NOT NULL,
  Dnumber         INT             NOT NULL,
  Mgr_ssn        CHAR(9)         NOT NULL,
  Mgr_start_date DATE,
  PRIMARY KEY (Dnumber),
  UNIQUE (Dname),
  FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
( Dnumber         INT             NOT NULL,
  Dlocation       VARCHAR(15)      NOT NULL,
  PRIMARY KEY (Dnumber, Dlocation),
  FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
```

continued on next slide

SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 5.7 (Fig. 6.1)-continued

CREATE TABLE PROJECT

(Pname	VARCHAR(15)	NOT NULL,
Pnumber	INT	NOT NULL,
Plocation	VARCHAR(15),	
Dnum	INT	NOT NULL,

PRIMARY KEY (Pnumber),
UNIQUE (Pname),
FOREIGN KEY (Dnum) **REFERENCES** DEPARTMENT(Dnumber);

CREATE TABLE WORKS_ON

(Essn	CHAR(9)	NOT NULL,
Pno	INT	NOT NULL,
Hours	DECIMAL(3,1)	NOT NULL,

PRIMARY KEY (Essn, Pno),
FOREIGN KEY (Essn) **REFERENCES** EMPLOYEE(Ssn),
FOREIGN KEY (Pno) **REFERENCES** PROJECT(Pnumber);

CREATE TABLE DEPENDENT

(Essn	CHAR(9)	NOT NULL,
Dependent_name	VARCHAR(15)	NOT NULL,
Sex	CHAR,	
Bdate	DATE,	
Relationship	VARCHAR(8),	

PRIMARY KEY (Essn, Dependent_name),
FOREIGN KEY (Essn) **REFERENCES** EMPLOYEE(Ssn);

Attribute Data Types and Domains in SQL

- **Basic data types**
 - **Numeric data types**
 - Integer numbers: INTEGER, INT, and SMALLINT
 - Floating-point (real) numbers: FLOAT or REAL, and DOUBLE PRECISION
 - **Character-string data types**
 - Fixed length: CHAR (n), CHARACTER (n)
 - Varying length: VARCHAR (n), CHAR VARYING (n), CHARACTER VARYING (n)

Attribute Data Types and Domains in SQL (cont'd.)

- **Bit-string** data types
 - Fixed length: BIT (n)
 - Varying length: BIT VARYING (n)
- **Boolean** data type
 - Values of TRUE or FALSE or NULL
- **DATE** data type
 - Ten positions
 - Components are YEAR, MONTH, and DAY in the form YYYY-MM-DD
 - Multiple mapping functions available in RDBMSs to change date formats

Attribute Data Types and Domains in SQL (cont'd.)

- Additional data types
 - **Timestamp** data type
 - Includes the DATE and TIME fields
 - Plus a minimum of six positions for decimal fractions of seconds
 - Optional WITH TIME ZONE qualifier
 - **INTERVAL** data type
 - Specifies a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp
 - **DATE, TIME, Timestamp, INTERVAL** data types can be cast or converted to string formats for comparison.

Attribute Data Types and Domains in SQL (cont'd.)

■ Domain

- Name used with the attribute specification
- Makes it easier to change the data type for a domain that is used by numerous attributes
- Improves schema readability
- Example:
 - `CREATE DOMAIN SSN_TYPE AS CHAR(9);`

■ TYPE

- User Defined Types (UDTs) are supported for object-oriented applications. (See Ch.12) Uses the command: `CREATE TYPE`

Specifying Constraints in SQL

Basic constraints:

- Relational Model has 3 basic constraint types that are supported in SQL:
 - **Key constraint**: A primary key value cannot be duplicated
 - **Entity Integrity Constraint**: A primary key value cannot be null
 - **Referential integrity constraints** : The “foreign key” must have a value that is already present as a primary key, or may be null.

Specifying Attribute Constraints

Other Restrictions on attribute domains:

- Default value of an attribute

- **DEFAULT** <value>

- **NULL** is not permitted for a particular attribute
(NOT NULL)

- **CHECK** clause

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

Specifying Key and Referential Integrity Constraints

■ PRIMARY KEY clause

- Specifies one or more attributes that make up the primary key of a relation
- Dnumber INT PRIMARY KEY;

■ UNIQUE clause

- Specifies alternate (secondary) keys (called CANDIDATE keys in the relational model).
- Dname VARCHAR (15) UNIQUE;

Specifying Key and Referential Integrity Constraints (cont'd.)

- **FOREIGN KEY clause**
 - Default operation: reject update on violation
 - Attach **referential triggered action clause**
 - Options include SET NULL, CASCADE, and SET DEFAULT
 - Action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE
 - CASCADE option suitable for “relationship” relations

Giving Names to Constraints

- Using the Keyword **CONSTRAINT**
 - Name a constraint
 - Useful for later altering

Default attribute values and referential integrity triggered action specification (Fig. 6.2)

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

Specifying Constraints on Tuples Using CHECK

- Additional Constraints on individual tuples within a relation are also possible using CHECK
- CHECK clauses at the end of a CREATE TABLE statement
 - Apply to each tuple individually
 - `CHECK (Dept_create_date <= Mgr_start_date);`

Basic Retrieval Queries in SQL

- SELECT statement
 - One basic statement for retrieving information from a database
- SQL allows a table to have two or more tuples that are identical in all their attribute values
 - Unlike relational model (relational model is strictly set-theory based)
 - Multiset or bag behavior
 - Tuple-id may be used as a key

The SELECT-FROM-WHERE Structure of Basic SQL Queries

■ Basic form of the SELECT statement:

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

where

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

The SELECT-FROM-WHERE Structure of Basic SQL Queries (cont'd.)

- Logical comparison operators
 - $=$, $<$, \leq , $>$, \geq , and \neq
- **Projection attributes**
 - Attributes whose values are to be retrieved
- **Selection condition**
 - Boolean condition that must be true for any retrieved tuple. Selection conditions include join conditions (see Ch.8) when multiple relations are involved.

Basic Retrieval Queries

<u>Bdate</u>	<u>Address</u>
1965-01-09	731 Fondren, Houston, TX

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

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

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

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;

Basic Retrieval Queries (Contd.)

(c)

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

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

Q2:

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

Ambiguous Attribute Names

- Same name can be used for two (or more) attributes in different relations
 - As long as the attributes are in different relations
 - Must **qualify** the attribute name with the relation name to prevent ambiguity

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

Aliasing, and Renaming

- **Aliases or tuple variables**
 - Declare alternative relation names E and S to refer to the EMPLOYEE relation twice in a query:

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

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

  - Recommended practice to abbreviate names and to prefix same or similar attribute from multiple tables.

# Aliasing, Renaming and Tuple Variables (contd.)

- The attribute names can also be renamed

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

- Note that the relation EMPLOYEE now has a variable name E which corresponds to a tuple variable
- The “AS” may be dropped in most SQL implementations

# Unspecified WHERE Clause and Use of the Asterisk

- Missing WHERE clause
  - Indicates no condition on tuple selection
- Effect is a CROSS PRODUCT
  - Result is all possible tuple combinations (or the Algebra operation of Cartesian Product— see Ch.8)

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

**Q9:**    **SELECT**    Ssn  
          **FROM**      EMPLOYEE;

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

# Unspecified WHERE Clause and Use of the Asterisk (cont'd.)

- Specify an asterisk (\*)
  - Retrieve all the attribute values of the selected tuples
  - The \* can be prefixed by the relation name; e.g., EMPLOYEE \*

Q1C: **SELECT** \*  
**FROM** EMPLOYEE  
**WHERE** Dno=5;

Q1D: **SELECT** \*  
**FROM** EMPLOYEE, DEPARTMENT  
**WHERE** Dname='Research' **AND** Dno=Dnumber;

Q10A: **SELECT** \*  
**FROM** EMPLOYEE, DEPARTMENT;

# Tables as Sets in SQL

- SQL does not automatically eliminate duplicate tuples in query results
- For aggregate operations (See sec 7.1.7) duplicates must be accounted for
- Use the keyword **DISTINCT** in the SELECT clause
  - Only distinct tuples should remain in the result

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

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

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

# Tables as Sets in SQL (cont'd.)

- Set operations
  - **UNION, EXCEPT (difference), INTERSECT**
  - Corresponding multiset operations: UNION ALL, EXCEPT ALL, INTERSECT ALL)
  - Type compatibility is needed for these operations to be valid

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

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

# Substring Pattern Matching and Arithmetic Operators

- **LIKE** comparison operator
  - Used for string **pattern matching**
  - % replaces an arbitrary number of zero or more characters
  - underscore (\_) replaces a single character
  - Examples: **WHERE** Address **LIKE** '%Houston,TX%';
  - **WHERE** Ssn **LIKE** '\_ \_ 1 \_ \_ 8901';
- **BETWEEN** comparison operator

E.g., in Q14 :

**WHERE**(Salary **BETWEEN** 30000 **AND** 40000)  
**AND** Dno = 5;

# Arithmetic Operations

- Standard arithmetic operators:
  - Addition (+), subtraction (-), multiplication (\*), and division (/) may be included as a part of **SELECT**
- **Query 13.** Show the resulting salaries if every employee working on the ‘ProductX’ project is given a 10 percent raise.

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

# Ordering of Query Results

- Use **ORDER BY** clause
  - Keyword **DESC** to see result in a descending order of values
  - Keyword **ASC** to specify ascending order explicitly
  - Typically placed at the end of the query

```
ORDER BY D.Dname DESC, E.Lname ASC,
E.Fname ASC
```

# Basic SQL Retrieval Query Block

```
SELECT <attribute list>
FROM <table list>
[WHERE <condition>]
[ORDER BY <attribute list>];
```

# INSERT, DELETE, and UPDATE Statements in SQL

- Three commands used to modify the database:
  - INSERT, DELETE, and UPDATE
- INSERT typically inserts a tuple (row) in a relation (table)
- UPDATE may update a number of tuples (rows) in a relation (table) that satisfy the condition
- DELETE may also update a number of tuples (rows) in a relation (table) that satisfy the condition

# 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
- Constraints on data types are observed automatically
- Any integrity constraints as a part of the DDL specification are enforced

# The INSERT Command

- Specify the relation name and a list of values for the tuple. All values including nulls are supplied.

```
U1: INSERT INTO EMPLOYEE
 VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);
```

- The variation below inserts multiple tuples where a new table is loaded values from the result of a query.

```
U3B: INSERT INTO WORKS_ON_INFO (Emp_name, Proj_name,
 Hours_per_week)
 SELECT E.Lname, P.Pname, W.Hours
 FROM PROJECT P, WORKS_ON W, EMPLOYEE E
 WHERE P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

# BULK LOADING OF TABLES

- Another variation of **INSERT** is used for bulk-loading of several tuples into tables
- A new table TNEW can be created with the same attributes as T and using LIKE and DATA in the syntax, it can be loaded with entire data.
- EXAMPLE:

```
CREATE TABLE D5EMPS LIKE EMPLOYEE
 (SELECT E.*
 FROM EMPLOYEE AS E
 WHERE E.Dno=5)

WITH DATA;
```

# DELETE

- Removes tuples from a relation
  - Includes a WHERE-clause to select the tuples to be deleted
  - Referential integrity should be enforced
  - 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

# The DELETE Command

- Removes tuples from a relation
  - Includes a WHERE clause to select the tuples to be deleted. The number of tuples deleted will vary.

|             |                                    |                              |
|-------------|------------------------------------|------------------------------|
| <b>U4A:</b> | <b>DELETE FROM</b><br><b>WHERE</b> | EMPLOYEE<br>Lname='Brown';   |
| <b>U4B:</b> | <b>DELETE FROM</b><br><b>WHERE</b> | EMPLOYEE<br>Ssn='123456789'; |
| <b>U4C:</b> | <b>DELETE FROM</b><br><b>WHERE</b> | EMPLOYEE<br>Dno=5;           |
| <b>U4D:</b> | <b>DELETE FROM</b>                 | EMPLOYEE;                    |

# 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 specified as part of DDL specification is enforced

# UPDATE (contd.)

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

```
U5: UPDATE PROJECT
 SET PLOCATION = 'Bellaire',
 DNUM = 5
 WHERE PNUMBER=10
```

# UPDATE (contd.)

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

```
U6:UPDATE EMPLOYEE
 SET SALARY = SALARY *1.1
 WHERE 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

# Additional Features of SQL

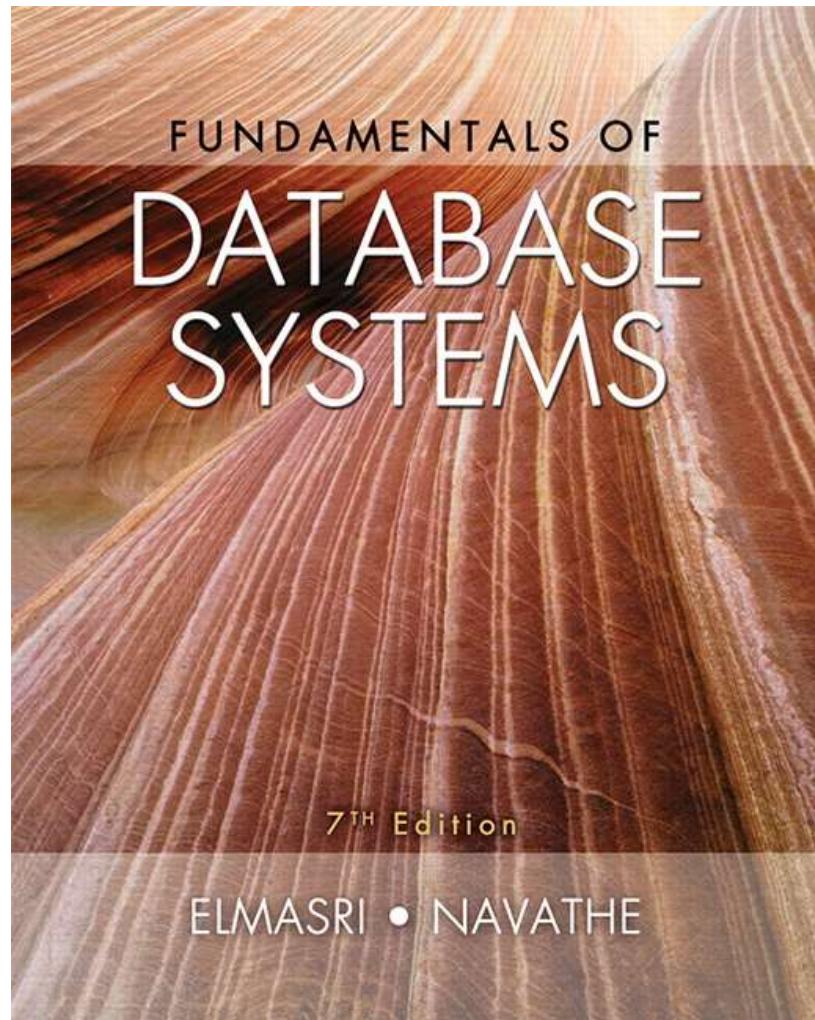
- Techniques for specifying complex retrieval queries (see Ch.7)
- Writing programs in various programming languages that include SQL statements: Embedded and dynamic SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC, SQL/PSM (Persistent Stored Module) (See Ch.10)
- Set of commands for specifying physical database design parameters, file structures for relations, and access paths, e.g., CREATE INDEX

# Additional Features of SQL (cont'd.)

- Transaction control commands (Ch.20)
- Specifying the granting and revoking of privileges to users (Ch.30)
- Constructs for creating triggers (Ch.26)
- Enhanced relational systems known as object-relational define relations as classes. Abstract data types (called User Defined Types- UDTs) are supported with CREATE TYPE
- New technologies such as XML (Ch.13) and OLAP (Ch.29) are added to versions of SQL

# Summary

- SQL
  - A Comprehensive language for relational database management
  - Data definition, queries, updates, constraint specification, and view definition
- Covered :
  - Data definition commands for creating tables
  - Commands for constraint specification
  - Simple retrieval queries
  - Database update commands



# **CHAPTER 7**

# **More SQL: Complex Queries, Triggers, Views, and Schema Modification**

# Chapter 7 Outline

- More Complex SQL Retrieval Queries
- Specifying Semantic Constraints as Assertions and Actions as Triggers
- Views (Virtual Tables) in SQL
- Schema Modification in SQL

# More Complex SQL Retrieval Queries

- Additional features allow users to specify more complex retrievals from database:
  - Nested queries, joined tables, and outer joins (in the FROM clause), aggregate functions, and grouping

# Comparisons Involving NULL and Three-Valued Logic

- Meanings of `NULL`
  - **Unknown value**
  - **Unavailable or withheld value**
  - **Not applicable attribute**
- Each individual `NULL` value considered to be different from every other `NULL` value
- SQL uses a three-valued logic:
  - `TRUE`, `FALSE`, and `UNKNOWN` (like `Maybe`)
- **`NULL = NULL` comparison is avoided**

# Comparisons Involving NULL and Three-Valued Logic (cont'd.)

**Table 7.1** Logical Connectives in Three-Valued Logic

| (a) | AND     | TRUE    | FALSE   | UNKNOWN |
|-----|---------|---------|---------|---------|
|     | TRUE    | TRUE    | FALSE   | UNKNOWN |
|     | FALSE   | FALSE   | FALSE   | FALSE   |
|     | UNKNOWN | UNKNOWN | FALSE   | UNKNOWN |
| (b) | OR      | TRUE    | FALSE   | UNKNOWN |
|     | TRUE    | TRUE    | TRUE    | TRUE    |
|     | FALSE   | TRUE    | FALSE   | UNKNOWN |
|     | UNKNOWN | TRUE    | UNKNOWN | UNKNOWN |
| (c) | NOT     |         |         |         |
|     | TRUE    | FALSE   |         |         |
|     | FALSE   | TRUE    |         |         |
|     | UNKNOWN | UNKNOWN |         |         |

# Comparisons Involving NULL and Three-Valued Logic (cont'd.)

- SQL allows queries that check whether an attribute value is NULL
  - IS or IS NOT NULL

**Query 18.** Retrieve the names of all employees who do not have supervisors.

**Q18:**    **SELECT**      Fname, Lname  
              **FROM**        EMPLOYEE  
              **WHERE**      Super\_ssn **IS** NULL;

# Nested Queries, Tuples, and Set/Multiset Comparisons

- Nested queries
  - Complete select-from-where blocks within WHERE clause of another query
  - **Outer query and nested subqueries**
- Comparison operator `IN`
  - Compares value  $v$  with a set (or multiset) of values  $V$
  - Evaluates to `TRUE` if  $v$  is one of the elements in  $V$

# Nested Queries (cont'd.)

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

# Nested Queries (cont'd.)

- Use tuples of values in comparisons
  - Place them within parentheses

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

# Nested Queries (cont'd.)

- Use other comparison operators to compare a single value  $v$ 
  - $= ANY$  (or  $= SOME$ ) operator
    - Returns TRUE if the value  $v$  is equal to some value in the set  $V$  and is hence equivalent to  $IN$
  - Other operators that can be combined with ANY (or SOME):  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ , and  $\neq$
  - ALL: value must exceed all values from nested query

|                                      |                                          |                           |                                |
|--------------------------------------|------------------------------------------|---------------------------|--------------------------------|
| <pre>SELECT<br/>FROM<br/>WHERE</pre> | Lname, Fname<br>EMPLOYEE<br>Salary > ALL | ( SELECT<br>FROM<br>WHERE | Salary<br>EMPLOYEE<br>Dno=5 ); |
|--------------------------------------|------------------------------------------|---------------------------|--------------------------------|

# Nested Queries (cont'd.)

- Avoid potential errors and ambiguities
  - Create tuple variables (aliases) for all tables referenced in SQL query

**Query 16.** Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16: SELECT E.Fname, E.Lname
 FROM EMPLOYEE AS E
 WHERE E.Ssn IN (SELECT Essn
 FROM DEPENDENT AS D
 WHERE E.Fname=D.Dependent_name
 AND E.Sex=D.Sex);
```

# Correlated Nested Queries

# The EXISTS and UNIQUE Functions in SQL for correlating queries

- EXISTS function
  - Check whether the result of a correlated nested query is empty or not. They are Boolean functions that return a TRUE or FALSE result.
- EXISTS and NOT EXISTS
  - Typically used in conjunction with a correlated nested query
- SQL function UNIQUE (Q)
  - Returns TRUE if there are no duplicate tuples in the result of query Q

# USE of EXISTS

Q7:

```
SELECT Fname, Lname
FROM Employee
WHERE EXISTS (SELECT *
 FROM DEPENDENT
 WHERE Ssn= Essn)

AND EXISTS (SELECT *
 FROM Department
 WHERE Ssn= Mgr_Ssn)
```

# USE OF NOT EXISTS

To achieve the “for all” (universal quantifier- see Ch.8) effect, we use double negation this way in SQL:

Query: List first and last name of employees who work on ALL projects controlled by Dno=5.

```
SELECT Fname, Lname
FROM Employee
WHERE NOT EXISTS ((SELECT Pnumber
 FROM PROJECT
 WHERE Dno=5)
```

```
EXCEPT (SELECT Pno
 FROM WORKS_ON
 WHERE Ssn= ESSn)
```

The above is equivalent to double negation: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

# Double Negation to accomplish “for all” in SQL

- **Q3B:**

```
SELECT Lname, Fname
 FROM EMPLOYEE
 WHERE NOT EXISTS (
 SELECT *
 FROM WORKS_ON B
 WHERE (B.Pno IN (SELECT Pnumber
 FROM PROJECT
 WHERE Dnum=5
 AND
 NOT EXISTS (SELECT *
 FROM WORKS_ON C
 WHERE C.Essn=Ssn
 AND C.Pno=B.Pno)));
```

The above is a direct rendering of: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

# Explicit Sets and Renaming of Attributes in SQL

- Can use explicit set of values in WHERE clause

Q17:      **SELECT**            **DISTINCT** Essn  
                **FROM**                WORKS\_ON  
                **WHERE**                Pno **IN** (1, 2, 3);

- Use qualifier AS followed by desired new name
  - Rename any attribute that appears in the result of a query

Q8A:      **SELECT**            E.Lname **AS** Employee\_name, S.Lname **AS** Supervisor\_name  
                **FROM**                EMPLOYEE **AS** E, EMPLOYEE **AS** S  
                **WHERE**                E.Super\_ssn=S.Ssn;

# Specifying Joined Tables in the FROM Clause of SQL

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query
- **The FROM clause in Q1A**
  - Contains a single joined table. JOIN may also be called INNER JOIN

**Q1A:**    **SELECT**      Fname, Lname, Address  
              **FROM**        (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)  
              **WHERE**      Dname='Research';

# Different Types of JOINed Tables in SQL

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN (LEFT, RIGHT, FULL )
- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Is equivalent to an implicit EQUIJOIN condition for each pair of attributes with same name from R and S

# NATURAL JOIN

- Rename attributes of one relation so it can be joined with another using NATURAL JOIN:

**Q1B:**      **SELECT**      Fname, Lname, Address  
                **FROM**        (EMPLOYEE **NATURAL JOIN**  
                          (DEPARTMENT AS DEPT (Dname, Dno, Mssn,  
                          Msdate)))  
                **WHERE**        Dname='Research';

The above works with EMPLOYEE.Dno = DEPT.Dno as an implicit join condition

# INNER and OUTER Joins

- INNER JOIN (**versus** OUTER JOIN)
  - Default type of join in a joined table
  - Tuple is included in the result only if a matching tuple exists in the other relation
- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table
- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of left table

# Example: LEFT OUTER JOIN

```
SELECT E.Lname AS Employee_Name
 S.Lname AS Supervisor_Name

FROM Employee AS E LEFT OUTER JOIN EMPLOYEE AS S
 ON E.Super_ssn = S.Ssn)
```

## ALTERNATE SYNTAX:

```
SELECT E.Lname , S.Lname
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.Super_ssn + = S.Ssn
```

# Multiway JOIN in the FROM clause

- FULL OUTER JOIN – combines result if LEFT and RIGHT OUTER JOIN
- Can nest JOIN specifications for a multiway join:

**Q2A:**    **SELECT** Pnumber, Dnum, Lname, Address, Bdate  
          **FROM** ((**PROJECT JOIN DEPARTMENT ON**  
                  Dnum=Dnumber) **JOIN EMPLOYEE ON**  
                  Mgr\_ssn=Ssn)  
          **WHERE** Plocation='Stafford';

# Aggregate Functions in SQL

- Used to summarize information from multiple tuples into a single-tuple summary
- Built-in aggregate functions
  - **COUNT, SUM, MAX, MIN, and AVG**
- **Grouping**
  - Create subgroups of tuples before summarizing
- To select entire groups, **HAVING clause** is used
- Aggregate functions can be used in the **SELECT clause** or in a **HAVING clause**

# Renaming Results of Aggregation

- Following query returns a single row of computed values from EMPLOYEE table:

```
Q19: SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG
 (Salary)
 FROM EMPLOYEE;
```

- The result can be presented with new names:

```
Q19A: SELECT SUM (Salary) AS Total_Sal, MAX (Salary) AS
 Highest_Sal, MIN (Salary) AS Lowest_Sal, AVG
 (Salary) AS Average_Sal
 FROM EMPLOYEE;
```

# Aggregate Functions in SQL (cont'd.)

- NULL values are discarded when aggregate functions are applied to a particular column

**Query 20.** Find the sum of the salaries of all employees of the ‘Research’ department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20: SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
 FROM (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
 WHERE Dname='Research';
```

**Queries 21 and 22.** Retrieve the total number of employees in the company (Q21) and the number of employees in the ‘Research’ department (Q22).

```
Q21: SELECT COUNT (*)
 FROM EMPLOYEE;
```

```
Q22: SELECT COUNT (*)
 FROM EMPLOYEE, DEPARTMENT
 WHERE DNO=DNUMBER AND DNAME='Research';
```

# Aggregate Functions on Booleans

- SOME and ALL may be applied as functions on Boolean Values.
- SOME returns true if at least one element in the collection is TRUE (similar to OR)
- ALL returns true if all of the elements in the collection are TRUE (similar to AND)

# Grouping: The GROUP BY Clause

- Partition relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently
- **GROUP BY** clause
  - Specifies grouping attributes
- COUNT (\*) counts the number of rows in the group

# Examples of GROUP BY

- The grouping attribute must appear in the SELECT clause:

Q24:      **SELECT**                    Dno, **COUNT (\*)**, **AVG** (Salary)  
              **FROM**                        EMPLOYEE  
              **GROUP BY**                  Dno;

- If the grouping attribute has NULL as a possible value, then a separate group is created for the null value (e.g., null Dno in the above query)
- GROUP BY may be applied to the result of a JOIN:

Q25:      **SELECT**                    Pnumber, Pname, **COUNT (\*)**  
              **FROM**                        PROJECT, WORKS\_ON  
              **WHERE**                        Pnumber=Pno  
              **GROUP BY**                  Pnumber, Pname;

# Grouping: The GROUP BY and HAVING Clauses (cont'd.)

## ■ **HAVING clause**

- Provides a condition to select or reject an entire group:
- **Query 26.** For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

Q26:      **SELECT**                    Pnumber, Pname, **COUNT (\*)**  
                **FROM**                        PROJECT, WORKS\_ON  
                **WHERE**                        Pnumber=Pno  
                **GROUP BY**                    Pnumber, Pname  
                **HAVING**                        **COUNT (\*) > 2;**

# Combining the WHERE and the HAVING Clause

- Consider the query: we want to count the *total* number of employees whose salaries exceed \$40,000 in each department, but only for departments where more than five employees work.
- INCORRECT QUERY:

```
SELECT Dno, COUNT (*)
FROM EMPLOYEE
WHERE Salary>40000
GROUP BY Dno
HAVING COUNT (*) > 5;
```

# Combining the WHERE and the HAVING Clause (continued)

## Correct Specification of the Query:

- Note: the WHERE clause applies tuple by tuple whereas HAVING applies to entire group of tuples

**Query 28.** For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than \$40,000.

Q28:    **SELECT**      Dnumber, COUNT (\*)  
          **FROM**        DEPARTMENT, EMPLOYEE  
          **WHERE**      Dnumber=Dno AND Salary>40000 AND  
                  ( **SELECT**        Dno  
                  **FROM**        EMPLOYEE  
                  **GROUP BY** Dno  
                  **HAVING**      COUNT (\*) > 5)

# Use of WITH

- The WITH clause allows a user to define a table that will only be used in a particular query (not available in all SQL implementations)
- Used for convenience to create a temporary “View” and use that immediately in a query
- Allows a more straightforward way of looking a step-by-step query

# Example of WITH

- See an alternate approach to doing Q28:

- Q28':  

```
WITH BIGDEPTS (Dno) AS
(SELECT Dno
 FROM EMPLOYEE
 GROUP BY Dno
 HAVING COUNT (*) > 5)
SELECT Dno, COUNT (*)
FROM EMPLOYEE
WHERE Salary>40000 AND Dno IN BIGDEPTS
GROUP BY Dno;
```

# Use of CASE

- SQL also has a CASE construct
- Used when a value can be different based on certain conditions.
- Can be used in any part of an SQL query where a value is expected
- Applicable when querying, inserting or updating tuples

# EXAMPLE of use of CASE

- The following example shows that employees are receiving different raises in different departments  
(A variation of the update U6)

- U6':

|               |                                               |
|---------------|-----------------------------------------------|
| <b>UPDATE</b> | <b>EMPLOYEE</b>                               |
| <b>SET</b>    | <b>Salary =</b>                               |
| <b>CASE</b>   | <b>WHEN Dno = 5 THEN</b> <b>Salary + 2000</b> |
|               | <b>WHEN Dno = 4 THEN</b> <b>Salary + 1500</b> |
|               | <b>WHEN Dno = 1 THEN</b> <b>Salary + 3000</b> |

# Recursive Queries in SQL

- An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor.
- This relationship is described by the foreign key Super\_ssn of the EMPLOYEE relation
- An example of a **recursive operation** is to retrieve all supervisees of a supervisory employee e at all levels—that is, all employees e' directly supervised by e, all employees e'' directly supervised by each employee e', all employees e''' directly supervised by each employee e'', and so on. Thus the CEO would have each employee in the company as a supervisee in the resulting table. Example shows such table SUP\_EMP with 2 columns (Supervisor,Supervisee(any level)):

# An EXAMPLE of RECURSIVE Query

- Q29: WITH RECURSIVE SUP\_EMP (SupSsn, EmpSsn) AS

```
 SELECT SupervisorSsn, Ssn
 FROM EMPLOYEE
 UNION
 SELECT E.Ssn, S.SupSsn
 FROM EMPLOYEE AS E, SUP_EMP AS S
 WHERE E.SupervisorSsn = S.EmpSsn)
 SELECT *
 FROM SUP_EMP;
```

- The above query starts with an empty SUP\_EMP and successively builds SUP\_EMP table by computing immediate supervisees first, then second level supervisees, etc. until a **fixed point** is reached and no more supervisees can be added

# EXPANDED Block Structure of SQL Queries

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

# Specifying Constraints as Assertions and Actions as Triggers

- Semantic Constraints: The following are beyond the scope of the EER and relational model
- **CREATE ASSERTION**
  - Specify additional types of constraints outside scope of built-in relational model constraints
- **CREATE TRIGGER**
  - Specify automatic actions that database system will perform when certain events and conditions occur

# Specifying General Constraints as Assertions in SQL

## ■ CREATE ASSERTION

- Specify a query that selects any tuples that violate the desired condition
- Use only in cases where it goes beyond a simple CHECK which applies to individual attributes and domains

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

# Introduction to Triggers in SQL

- CREATE TRIGGER statement
  - Used to monitor the database
- Typical trigger has three components which make it a rule for an “active database” (more on active databases in section 26.1) :
  - **Event(s)**
  - **Condition**
  - **Action**

# USE OF TRIGGERS

- AN EXAMPLE with standard Syntax.(Note : other SQL implementations like PostgreSQL use a different syntax.)

R5:

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn ON
EMPLOYEE
```

**FOR EACH ROW**

```
WHEN (NEW.Salary > (SELECT Salary FROM EMPLOYEE
 WHERE Ssn = NEW. Supervisor_Ssn))
INFORM_SUPERVISOR (NEW.Supervisor.Ssn, New.Ssn)
```

# Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables called the **defining tables**
  - Considered to be a virtual table that is not necessarily populated

# Specification of Views in SQL

## ■ CREATE VIEW command

- Give table name, list of attribute names, and a query to specify the contents of the view
- In V1, attributes retain the names from base tables. In V2, attributes are assigned names

|     |             |                                             |
|-----|-------------|---------------------------------------------|
| V1: | CREATE VIEW | WORKS_ON1                                   |
|     | AS SELECT   | Fname, Lname, Pname, Hours                  |
|     | FROM        | EMPLOYEE, PROJECT, WORKS_ON                 |
|     | WHERE       | Ssn=Essn AND Pno=Pnumber;                   |
| V2: | CREATE VIEW | DEPT_INFO(Dept_name, No_of_emps, Total_sal) |
|     | AS SELECT   | Dname, COUNT (*), SUM (Salary)              |
|     | FROM        | DEPARTMENT, EMPLOYEE                        |
|     | WHERE       | Dnumber=Dno                                 |
|     | GROUP BY    | Dname;                                      |

# Specification of Views in SQL (cont'd.)

- Once a View is defined, SQL queries can use the View relation in the FROM clause
- View is always up-to-date
  - Responsibility of the DBMS and not the user
- **DROP VIEW** command
  - Dispose of a view

# View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Strategy1: Query modification** approach
  - Compute the view as and when needed. Do not store permanently
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute

# View Materialization

## ■ **Strategy 2: View materialization**

- Physically create a temporary view table when the view is first queried
- Keep that table on the assumption that other queries on the view will follow
- Requires efficient strategy for automatically updating the view table when the base tables are updated

## ■ **Incremental update strategy for materialized views**

- DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table

# View Materialization (contd.)

- Multiple ways to handle materialization:
  - **immediate update** strategy updates a view as soon as the base tables are changed
  - **lazy update** strategy updates the view when needed by a view query
  - **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date). This is commonly used in Banks, Retail store operations, etc.

# View Update

- Update on a view defined on a single table without any aggregate functions
  - Can be mapped to an update on underlying base table- possible if the primary key is preserved in the view
- Update not permitted on aggregate views. E.g.,

|             |               |                   |
|-------------|---------------|-------------------|
| <b>UV2:</b> | <b>UPDATE</b> | <b>DEPT_INFO</b>  |
|             | <b>SET</b>    | Total_sal=100000  |
|             | <b>WHERE</b>  | Dname='Research'; |

cannot be processed because Total\_sal is a computed value in the view definition

# View Update and Inline Views

- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended
- Clause **WITH CHECK OPTION**
  - Must be added at the end of the view definition if a view is to be updated to make sure that tuples being updated stay in the view
- In-line view
  - Defined in the `FROM` clause of an SQL query (e.g., we saw its used in the `WITH` example)

# Views as authorization mechanism

- SQL query authorization statements (GRANT and REVOKE) are described in detail in Chapter 30
- Views can be used to hide certain attributes or tuples from unauthorized users
- E.g., For a user who is only allowed to see employee information for those who work for department 5, he may only access the view

**DEPT5EMP:**

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

# Schema Change Statements in SQL

## ■ Schema evolution commands

- DBA may want to change the schema while the database is operational
- Does not require recompilation of the database schema

# The DROP Command

- **DROP command**
  - Used to drop named schema elements, such as tables, domains, or constraint
- **Drop behavior options:**
  - CASCADE and RESTRICT
- **Example:**
  - `DROP SCHEMA COMPANY CASCADE;`
  - This removes the schema and all its elements including tables, views, constraints, etc.

# The ALTER table command

- **Alter table actions** include:
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints
- **Example:**
  - `ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);`

# Adding and Dropping Constraints

- Change constraints specified on a table
  - Add or drop a named constraint

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

# Dropping Columns, Default Values

- To drop a column
    - Choose either CASCADE or RESTRICT
    - CASCADE would drop the column from views etc.
    - RESTRICT is possible if no views refer to it.
- ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN  
Address CASCADE;**

- Default values can be dropped and altered :  
**ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr\_ssn  
DROP DEFAULT;**  
**ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr\_ssn SET  
DEFAULT '333445555';**

# Table 7.2 Summary of SQL Syntax

**Table 7.2** Summary of SQL Syntax

---

```
CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
 { , <column name> <column type> [<attribute constraint>] }
 [<table constraint> { , <table constraint> }])
```

---

```
DROP TABLE <table name>
```

```
ALTER TABLE <table name> ADD <column name> <column type>
```

---

```
SELECT [DISTINCT] <attribute list>
```

```
FROM (<table name> { <alias> } | <joined table>) { , (<table name> { <alias> } | <joined table>) }
[WHERE <condition>]
```

```
[GROUP BY <grouping attributes> [HAVING <group selection condition>]]
```

```
[ORDER BY <column name> [<order>] { , <column name> [<order>] }]
```

---

```
<attribute list> ::= (* | (<column name> | <function> (([DISTINCT] <column name> | *))
 { , (<column name> | <function> (([DISTINCT] <column name> | *))))))
```

---

```
<grouping attributes> ::= <column name> { , <column name> }
```

---

```
<order> ::= (ASC | DESC)
```

---

```
INSERT INTO <table name> [(<column name> { , <column name> })]
(VALUES (<constant value> , { <constant value> }) { , (<constant value> { , <constant value> }) }
| <select statement>)
```

---

*continued on next slide*

# Table 7.2 (continued)

## Summary of SQL Syntax

**Table 7.2** Summary of SQL Syntax

---

DELETE FROM <table name>

[ WHERE <selection condition> ]

---

UPDATE <table name>

SET <column name> = <value expression> { , <column name> = <value expression> }

[ WHERE <selection condition> ]

---

CREATE [ UNIQUE] INDEX <index name>

ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )

[ CLUSTER ]

---

DROP INDEX <index name>

---

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]

AS <select statement>

---

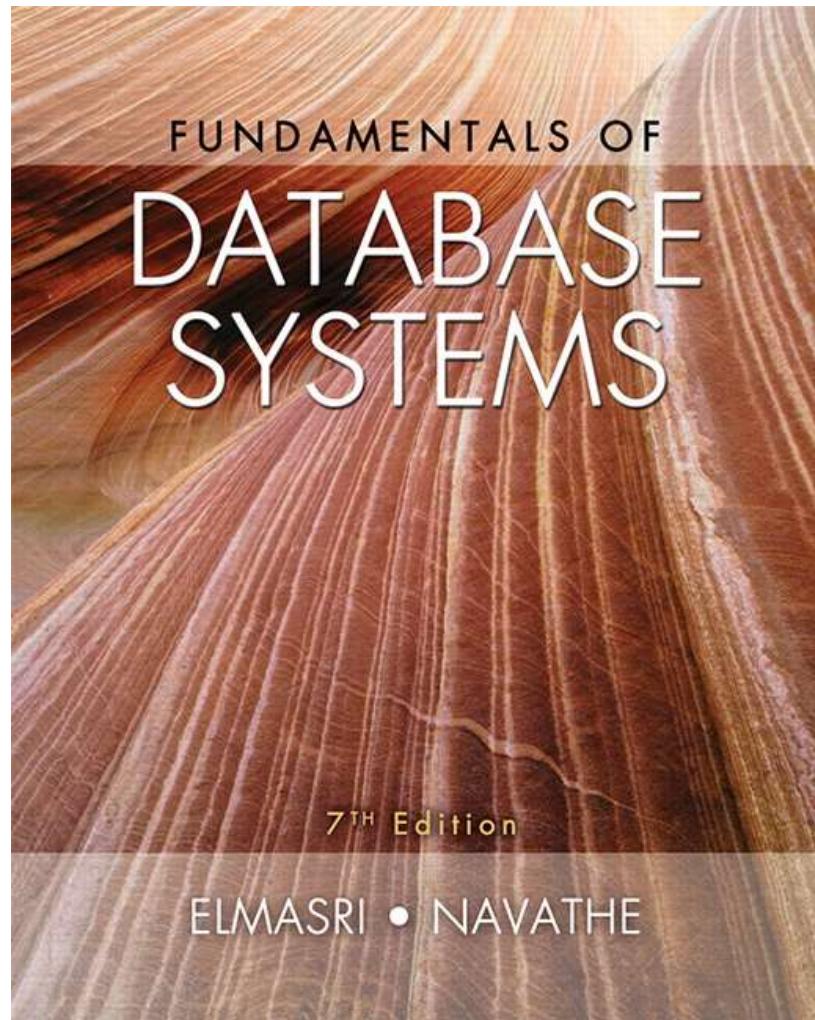
DROP VIEW <view name>

---

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

# Summary

- Complex SQL:
  - Nested queries, joined tables (in the FROM clause), outer joins, aggregate functions, grouping
- Handling semantic constraints with CREATE ASSERTION and CREATE TRIGGER
- CREATE VIEW statement and materialization strategies
- Schema Modification for the DBAs using ALTER TABLE , ADD and DROP COLUMN, ALTER CONSTRAINT etc .



# **CHAPTER 8**

## **The Relational Algebra and The Relational Calculus (plus QBE- Appendix C)**

# Chapter Outline

- 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
- Example Database Application (COMPANY)
- Overview of the QBE language (appendix D)

# Relational Algebra Overview

- Relational algebra is the basic set of operations for the relational model
- These operations enable a user to specify **basic retrieval requests** (or **queries**)
- The result of an operation is a *new relation*, which may have been formed from one or more *input* relations
  - This property makes the algebra “closed” (all objects in relational algebra are relations)

# Relational Algebra Overview (continued)

- The **algebra operations** thus produce new relations
  - These can be further manipulated using operations of the same algebra
- A sequence of relational algebra operations forms a **relational algebra expression**
  - The result of a relational algebra expression is also a relation that represents the result of a database query (or retrieval request)

# Brief History of Origins of Algebra

- Muhammad ibn Musa al-Khwarizmi (800-847 CE) – from Morocco wrote a book titled al-jabr about arithmetic of variables
  - Book was translated into Latin.
  - Its title (al-jabr) gave Algebra its name.
- Al-Khwarizmi called variables “shay”
  - “Shay” is Arabic for “thing”.
  - Spanish transliterated “shay” as “xay” (“x” was “sh” in Spain).
  - In time this word was abbreviated as x.
- Where does the word Algorithm come from?
  - Algorithm originates from “al-Khwarizmi”
  - Reference: PBS (<http://www.pbs.org/empires/islam/innoalgebra.html>)

# Relational Algebra Overview

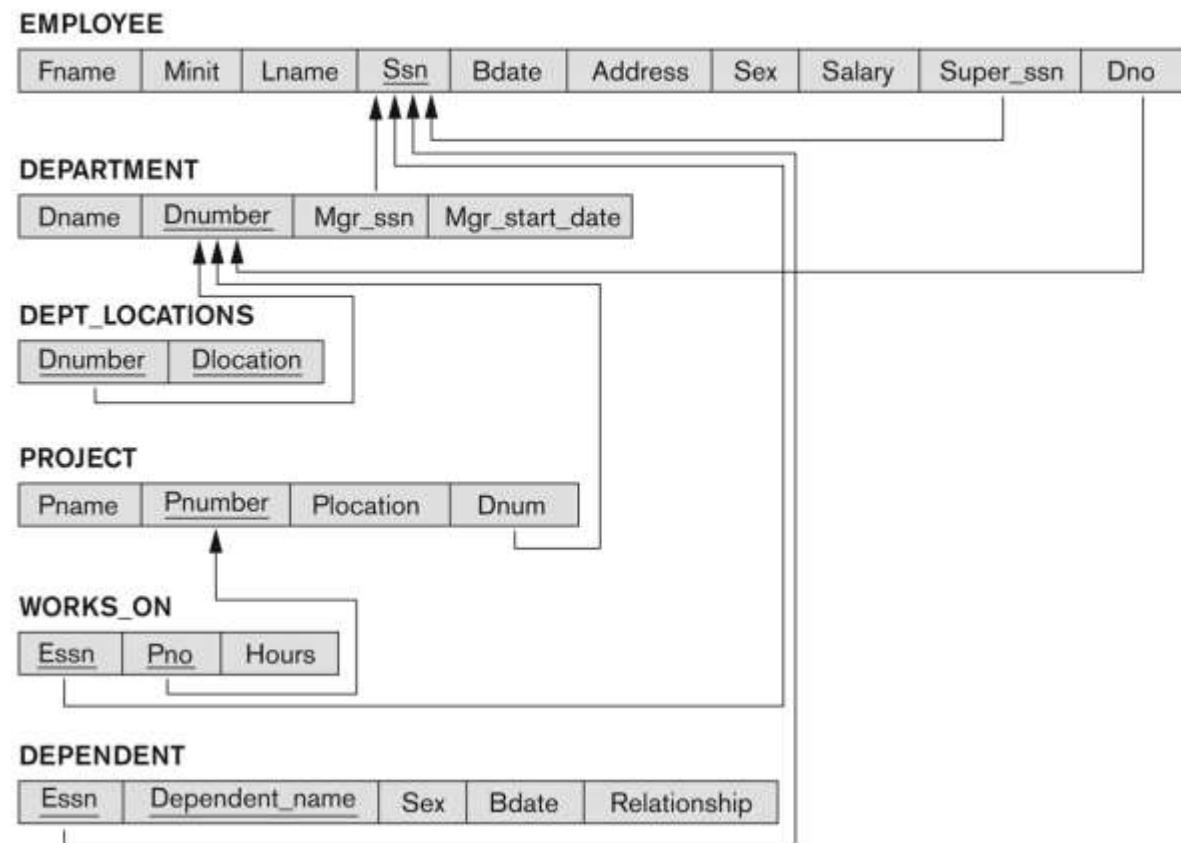
- Relational Algebra consists of several groups of operations
  - Unary Relational Operations
    - SELECT (symbol:  $\sigma$  (sigma))
    - PROJECT (symbol:  $\pi$  (pi))
    - RENAME (symbol:  $\rho$  (rho))
  - Relational Algebra Operations From Set Theory
    - UNION (  $\cup$  ), INTERSECTION (  $\cap$  ), DIFFERENCE (or MINUS,  $-$ )
    - CARTESIAN PRODUCT (  $\times$  )
  - Binary Relational Operations
    - JOIN (several variations of JOIN exist)
    - DIVISION
  - Additional Relational Operations
    - OUTER JOINS, OUTER UNION
    - AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, MAX)

# Database State for COMPANY

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

**Figure 5.7**

Referential integrity constraints displayed on the COMPANY relational database schema.



# Unary Relational Operations: SELECT

- The SELECT operation (denoted by  $\sigma$  (sigma)) is used to select a *subset* of the tuples from a relation based on a **selection condition**.
  - The selection condition acts as a **filter**
  - Keeps only those tuples that satisfy the qualifying condition
  - Tuples satisfying the condition are *selected* whereas the other tuples are discarded (*filtered out*)
- Examples:
  - Select the EMPLOYEE tuples whose department number is 4:  
$$\sigma_{DNO = 4} (\text{EMPLOYEE})$$
  - Select the employee tuples whose salary is greater than \$30,000:  
$$\sigma_{\text{SALARY} > 30,000} (\text{EMPLOYEE})$$

# Unary Relational Operations: SELECT

- In general, the *select* operation is denoted by  
 $\sigma_{<\text{selection condition}>}(R)$  where
  - the symbol  $\sigma$  (sigma) is used to denote the *select* operator
  - the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
  - tuples that make the condition **true** are selected
    - appear in the result of the operation
  - tuples that make the condition **false** are filtered out
    - discarded from the result of the operation

# Unary Relational Operations: SELECT (continued)

## ■ SELECT Operation Properties

- The SELECT operation  $\sigma_{<\text{selection condition}>} (R)$  produces a relation S that has the same schema (same attributes) as R
- SELECT  $\sigma$  is commutative:
  - $\sigma_{<\text{condition1}>} (\sigma_{<\text{condition2}>} (R)) = \sigma_{<\text{condition2}>} (\sigma_{<\text{condition1}>} (R))$
- Because of commutativity property, a cascade (sequence) of SELECT operations may be applied in any order:
  - $\sigma_{<\text{cond1}>} (\sigma_{<\text{cond2}>} (\sigma_{<\text{cond3}>} (R))) = \sigma_{<\text{cond2}>} (\sigma_{<\text{cond3}>} (\sigma_{<\text{cond1}>} (R)))$
- A cascade of SELECT operations may be replaced by a single selection with a conjunction of all the conditions:
  - $\sigma_{<\text{cond1}>} (\sigma_{<\text{cond2}>} (\sigma_{<\text{cond3}>} (R))) = \sigma_{<\text{cond1}> \text{ AND } <\text{cond2}> \text{ AND } <\text{cond3}>} (R))$
- The number of tuples in the result of a SELECT is less than (or equal to) the number of tuples in the input relation R

# The following query results refer to this database state

**Figure 5.6**

One possible database state for the COMPANY relational database schema.

**EMPLOYEE**

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

**DEPARTMENT**

| Dname          | Dnumber | Mgr_ssn   | Mgr_start_date |
|----------------|---------|-----------|----------------|
| Research       | 5       | 333445555 | 1988-05-22     |
| Administration | 4       | 987654321 | 1995-01-01     |
| Headquarters   | 1       | 888665555 | 1981-08-19     |

**DEPT\_LOCATIONS**

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

**WORKS\_ON**

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

**PROJECT**

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

**DEPENDENT**

| Essn      | Dependent_name | Sex | Bdate      | Relationship |
|-----------|----------------|-----|------------|--------------|
| 333445555 | Alice          | F   | 1986-04-05 | Daughter     |
| 333445555 | Theodore       | M   | 1983-10-25 | Son          |
| 333445555 | Joy            | F   | 1958-05-03 | Spouse       |
| 987654321 | Abner          | M   | 1942-02-28 | Spouse       |
| 123456789 | Michael        | M   | 1988-01-04 | Son          |
| 123456789 | Alice          | F   | 1988-12-30 | Daughter     |
| 123456789 | Elizabeth      | F   | 1987-05-05 | Spouse       |

# Unary Relational Operations: PROJECT

- PROJECT Operation is denoted by  $\pi$  (pi)
- This operation keeps certain *columns* (attributes) from a relation and discards the other columns.
  - PROJECT creates a vertical partitioning
    - The list of specified columns (attributes) is kept in each tuple
    - The other attributes in each tuple are discarded
- Example: To list each employee's first and last name and salary, the following is used:

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

# Unary Relational Operations: PROJECT (cont.)

- The general form of the *project* operation is:

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

- $\pi$  (pi) is the symbol used to represent the *project* operation
- $\langle \text{attribute list} \rangle$  is the desired list of attributes from relation R.
- The project operation *removes any duplicate tuples*
  - This is because the result of the *project* operation must be a *set of tuples*
    - Mathematical sets *do not allow* duplicate elements.

# Unary Relational Operations: PROJECT (contd.)

- PROJECT Operation Properties
  - The number of tuples in the result of projection  $\pi_{<\text{list}>}(R)$  is always less or equal to the number of tuples in R
    - If the list of attributes includes a key of R, then the number of tuples in the result of PROJECT is equal to the number of tuples in R
  - PROJECT is *not* commutative
    - $\pi_{<\text{list1}>}(\pi_{<\text{list2}>}(R)) = \pi_{<\text{list1}>}(R)$  as long as  $<\text{list2}>$  contains the attributes in  $<\text{list1}>$

# Examples of applying SELECT and PROJECT operations

Figure 8.1 Results of SELECT and PROJECT operations. (a)  $\sigma_{Dno=4 \text{ AND } Salary > 25000} \text{ OR } (Dno=5 \text{ AND } Salary > 30000)$  (EMPLOYEE). (b)  $\pi_{Lname, Fname, Salary}$ (EMPLOYEE). (c)  $\pi_{Sex, Salary}$ (EMPLOYEE).

(a)

| Fname    | Minit | Lname   | Ssn       | Bdate      | Address                  | Sex | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|------------|--------------------------|-----|--------|-----------|-----|
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    | M   | 40000  | 888665555 | 5   |
| 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   |

(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  |

# Relational Algebra Expressions

- 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.

# Single expression versus sequence of relational operations (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: RENAME

- The RENAME operator is denoted by  $\rho$  (rho)
- In some cases, we may want to *rename* the attributes of a relation or the relation name or both
  - Useful when a query requires multiple operations
  - Necessary in some cases (see JOIN operation later)

# Unary Relational Operations: RENAME (continued)

- The general RENAME operation  $\rho$  can be expressed by any of the following forms:
  - $\rho_{S(B_1, B_2, \dots, B_n)}(R)$  changes both:
    - the relation name to  $S$ , and
    - the column (attribute) names to  $B_1, B_1, \dots, B_n$
  - $\rho_S(R)$  changes:
    - the *relation name* only to  $S$
  - $\rho_{(B_1, B_2, \dots, B_n)}(R)$  changes:
    - the *column (attribute) names* only to  $B_1, B_1, \dots, B_n$

# Unary Relational Operations: RENAME (continued)

- For convenience, we also use a *shorthand* for renaming attributes in an intermediate relation:
  - If we write:
    - $\text{RESULT} \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{SALARY}}(\text{DEP5_EMPS})$
    - $\text{RESULT}$  will have the *same attribute names* as  $\text{DEP5_EMPS}$  (same attributes as  $\text{EMPLOYEE}$ )
  - If we write:
    - $\text{RESULT}(F, M, L, S, B, A, SX, SAL, SU, DNO) \leftarrow \rho_{\text{RESULT}(F.M.L.S.B,A,SX,SAL,SU,DNO)}(\text{DEP5_EMPS})$
    - The 10 attributes of  $\text{DEP5_EMPS}$  are *renamed* to  $F, M, L, S, B, A, SX, SAL, SU, DNO$ , respectively

**Note:** the  $\leftarrow$  symbol is an assignment operator

# Example of applying multiple operations and RENAME

**Figure 8.2** Results of a sequence of operations. (a)  $\pi_{\text{Fname}, \text{Lname}, \text{Salary}}(\sigma_{\text{Dno}=3}(\text{EMPLOYEE}))$ .  
(b) 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 | Super_ssn | Dno |
|----------|-------|---------|-----------|------------|--------------------------|-----|--------|-----------|-----|
| John     | B     | Smith   | 123456789 | 1985-01-09 | 731 Fendren, Houston, TX | M   | 30000  | 333445555 | 5   |
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    | M   | 40000  | 888665555 | 5   |
| Ramesh   | K     | Narayan | 666884444 | 1982-09-15 | 975 Fire Oak, Humble, TX | M   | 38000  | 333445555 | 5   |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX   | F   | 25000  | 333445555 | 5   |

R

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

# Relational Algebra Operations from Set Theory: UNION

## ■ UNION Operation

- Binary operation, denoted by  $\cup$
- The result of  $R \cup S$ , 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
- The two operand relations  $R$  and  $S$  must be “type compatible” (or UNION compatible)
  - $R$  and  $S$  must have same number of attributes
  - Each pair of corresponding attributes must be type compatible (have same or compatible domains)

# Relational Algebra Operations from Set Theory: UNION

## ■ Example:

- To retrieve the social security numbers of all employees who either *work in department 5* (RESULT1 below) or *directly supervise an employee who works in department 5* (RESULT2 below)
- We can use the UNION operation as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$$
$$\text{RESULT1} \leftarrow \pi_{\text{SSN}}(\text{DEP5\_EMPS})$$
$$\text{RESULT2(SSN)} \leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5\_EMPS})$$
$$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$$

- The union operation produces the tuples that are in either RESULT1 or RESULT2 or both

## **Figure 8.3** Result of the UNION operation $\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$ .

**RESULT1**

| Ssn       |
|-----------|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |

**RESULT2**

| Ssn       |
|-----------|
| 333445555 |
| 888665555 |

**RESULT**

| Ssn       |
|-----------|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |
| 888665555 |

# Relational Algebra Operations from Set Theory

- Type Compatibility of operands is required for the binary set operation UNION  $\cup$ , (also for INTERSECTION  $\cap$ , and SET DIFFERENCE  $-$ , see next slides)
- R1(A1, A2, ..., An) and R2(B1, B2, ..., Bn) are type compatible if:
  - they have the same number of attributes, and
  - the domains of corresponding attributes are type compatible (i.e.  $\text{dom}(A_i) = \text{dom}(B_i)$  for  $i=1, 2, \dots, n$ ).
- The resulting relation for  $R1 \cup R2$  (also for  $R1 \cap R2$ , or  $R1 - R2$ , see next slides) has the same attribute names as the *first* operand relation R1 (by convention)

# Relational Algebra Operations from Set Theory: INTERSECTION

- INTERSECTION is denoted by  $\cap$
- The result of the operation  $R \cap S$ , is a relation that includes all tuples that are in both  $R$  and  $S$ 
  - The attribute names in the result will be the same as the attribute names in  $R$
- The two operand relations  $R$  and  $S$  must be “type compatible”

# Relational Algebra Operations from Set Theory: SET DIFFERENCE (cont.)

- SET DIFFERENCE (also called MINUS or EXCEPT) is denoted by –
- The result of  $R - S$ , is a relation that includes all tuples that are in R but not in S
  - The attribute names in the result will be the same as the attribute names in R
- The two operand relations R and S must be “type compatible”

# Example to illustrate the result of UNION, INTERSECT, and DIFFERENCE

**Figure 8.4** The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT  $\cup$  INSTRUCTOR. (c) STUDENT  $\cap$  INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) STUDENT

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

INSTRUCTOR

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

(b)

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

(c)

| Fn     | Ln   |
|--------|------|
| Susan  | Yao  |
| Ramesh | Shah |

(d)

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

(e)

| Fname   | Lname   |
|---------|---------|
| John    | Smith   |
| Ricardo | Browne  |
| Francis | Johnson |

# Some properties of UNION, INTERSECT, and DIFFERENCE

- Notice that both union and intersection are *commutative* operations; that is
  - $R \cup S = S \cup R$ , 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
  - $R \cup (S \cup T) = (R \cup S) \cup T$
  - $(R \cap S) \cap T = R \cap (S \cap T)$
- The minus operation is not commutative; that is, in general
  - $R - S \neq S - R$

# Relational Algebra Operations from Set Theory: CARTESIAN PRODUCT

- CARTESIAN (or CROSS) PRODUCT Operation
  - This operation is used to combine tuples from two relations in a combinatorial fashion.
  - Denoted by  $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$
  - Result 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 state 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"

# Relational Algebra Operations from Set Theory: CARTESIAN PRODUCT (cont.)

- Generally, CROSS PRODUCT is not a meaningful operation
  - Can become meaningful when followed by other operations
- Example (not meaningful):
  - $\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{SEX}='F'}(\text{EMPLOYEE})$
  - $\text{EMP NAMES} \leftarrow \pi_{\text{FNAME, LNAME, SSN}}(\text{FEMALE\_EMPS})$
  - $\text{EMP\_DEPENDENTS} \leftarrow \text{EMP NAMES} \times \text{DEPENDENT}$
- $\text{EMP\_DEPENDENTS}$  will contain every combination of  $\text{EMP NAMES}$  and  $\text{DEPENDENT}$ 
  - whether or not they are actually related

# Relational Algebra Operations from Set Theory: CARTESIAN PRODUCT (cont.)

- To keep only combinations where the DEPENDENT is related to the EMPLOYEE, we add a SELECT operation as follows
- Example (meaningful):
  - $\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{SEX}='F'}(\text{EMPLOYEE})$
  - $\text{EMPNAMES} \leftarrow \pi_{\text{FNAME, LNAME, SSN}}(\text{FEMALE\_EMPS})$
  - $\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$
  - $\text{ACTUAL\_DEPS} \leftarrow \sigma_{\text{SSN}=\text{ESSN}}(\text{EMP\_DEPENDENTS})$
  - $\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME, DEPENDENT\_NAME}}(\text{ACTUAL\_DEPS})$
- RESULT will now contain the name of female employees and their dependents

## Figure 8.5 The CARTESIAN PRODUCT (CROSS PRODUCT) operation.

FEMALE\_EMPS

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

EMPNAME

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

*continued on next slide*

# Figure 8.5 (continued) The CARTESIAN PRODUCT (CROSS PRODUCT) operation.

EMP\_DEPENDENTS

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

continued on next slide

## Figure 8.5 (continued) The CARTESIAN PRODUCT (CROSS PRODUCT) operation.

ACTUAL\_DEPENDENTS

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

RESULT

| Fname    | Lname   | Dependent_name |
|----------|---------|----------------|
| Jennifer | Wallace | Abner          |

# Binary Relational Operations: JOIN

- JOIN Operation (denoted by  $\bowtie$ )
  - 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 combines this sequence into a single operation
  - This operation is very important for any relational database with more than a single relation, because it allows us *combine related tuples* from various relations
  - The general form of a join operation on two relations R(A<sub>1</sub>, A<sub>2</sub>, . . . , A<sub>n</sub>) and S(B<sub>1</sub>, B<sub>2</sub>, . . . , B<sub>m</sub>) 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: JOIN (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  operation.
- DEPT\_MGR  $\leftarrow$  DEPARTMENT  MGRSSN=SSN EMPLOYEE
- MGRSSN=SSN is the join condition
  - Combines each department record with the employee who manages the department
  - The join condition can also be specified as DEPARTMENT.MGRSSN= EMPLOYEE.SSN

## Figure 8.6 Result of the JOIN operation

$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT}^{\text{IXI}} \text{ Mgr\_ssn} = \text{Ssn} \text{EMPLOYEE}.$

**DEPT\_MGR**

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

# Some properties of JOIN

- Consider the following JOIN operation:
  - $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$   
 $R.A_i=S.B_j$
  - Result 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 state has one tuple for each combination of tuples—r from R and s from S, but *only if they satisfy the join condition  $r[A_i]=s[B_j]$*
    - Hence, if R has  $n_R$  tuples, and S has  $n_S$  tuples, then the join result will generally have *less than  $n_R * n_S$*  tuples.
    - Only related tuples (based on the join condition) will appear in the result

# Some properties of JOIN

- The general case of JOIN operation is called a Theta-join: R  S  
*theta*
- The join condition is called *theta*
- *Theta* can be any general boolean expression on the attributes of R and S; for example:
  - $R.A_i < S.B_j \text{ AND } (R.A_k = S.B_l \text{ OR } R.A_p < S.B_q)$
- Most join conditions involve one or more equality conditions “AND”ed together; for example:
  - $R.A_i = S.B_j \text{ AND } R.A_k = S.B_l \text{ AND } R.A_p = S.B_q$

# Binary Relational Operations: EQUIJOIN

- 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 an EQUIJOIN.

# Binary Relational Operations: NATURAL JOIN Operation

- NATURAL JOIN Operation
  - Another variation of JOIN called NATURAL JOIN — denoted by \* — was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.
    - because one of each pair of attributes with identical values is superfluous
  - 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

## NATURAL JOIN (continued)

- Example: To apply a natural join on the DNUMBER attributes of DEPARTMENT and DEPT\_LOCATIONS, it is sufficient to write:
  - $\text{DEPT\_LOCS} \leftarrow \text{DEPARTMENT} * \text{DEPT\_LOCATIONS}$
- Only attribute with the same name is DNUMBER
- An implicit join condition is created based on this attribute:  
 $\text{DEPARTMENT.DNUMBER}=\text{DEPT\_LOCATIONS.DNUMBER}$
  
- Another example:  $Q \leftarrow R(A,B,C,D) * S(C,D,E)$ 
  - The implicit join condition includes *each pair* of attributes with the same name, “AND”ed together:
    - $R.C=S.C \text{ AND } R.D=S.D$
  - Result keeps only one attribute of each such pair:
    - $Q(A,B,C,D,E)$

# Example of NATURAL JOIN operation

Figure 8.7 Results of two natural join operations. (a) proj\_dept ← project \* dept. (b) dept\_locs ← department \* dept\_locations.

(a)

PROJ\_DEPT

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

(b)

DEPT\_LOCS

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

# Complete Set of Relational Operations

- The set of operations including SELECT  $\sigma$ , PROJECT  $\pi$ , UNION  $\cup$ , DIFFERENCE  $-$ , RENAME  $\rho$ , 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: DIVISION

- 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  $S$ .

# Example of DIVISION

Figure 8.8 The DIVISION operation. (a) Dividing SSN\_PNOS by SMITH\_PNOS. (b)  $T \leftarrow R \div S$ .

(a)

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

| SMITH_PNOS |  |
|------------|--|
| Pno        |  |
| 1          |  |
| 2          |  |

SSNS

| Ssn       |
|-----------|
| 123456789 |
| 453453453 |

(b)

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

| S  |  |
|----|--|
| A  |  |
| a1 |  |
| a2 |  |
| a3 |  |

| T  |  |
|----|--|
| B  |  |
| b1 |  |
| b4 |  |

# Table 8.1 Operations of Relational Algebra

**Table 8.1** 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.                                                                                            | $\pi_{\langle \text{attribute list} \rangle}(R)$                                                                                                                                              |
| 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<br>$R_1 \bowtie_{(\langle \text{join attributes 1} \rangle, \langle \text{join attributes 2} \rangle)} R_2$                    |
| 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. | $R_1 *_{\langle \text{join condition} \rangle} R_2$ ,<br>OR $R_1 *_{(\langle \text{join attributes 1} \rangle, \langle \text{join attributes 2} \rangle)} R_2$<br>$R_2 \text{ OR } R_1 * R_2$ |

*continued on next slide*

# Table 8.1 Operations of Relational Algebra (continued)

**Table 8.1** Operations of Relational Algebra

| OPERATION         | PURPOSE                                                                                                                                                            | NOTATION             |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|
| 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 $Z = X \cup Y$ . | $R_1(Z) \div R_2(Y)$ |

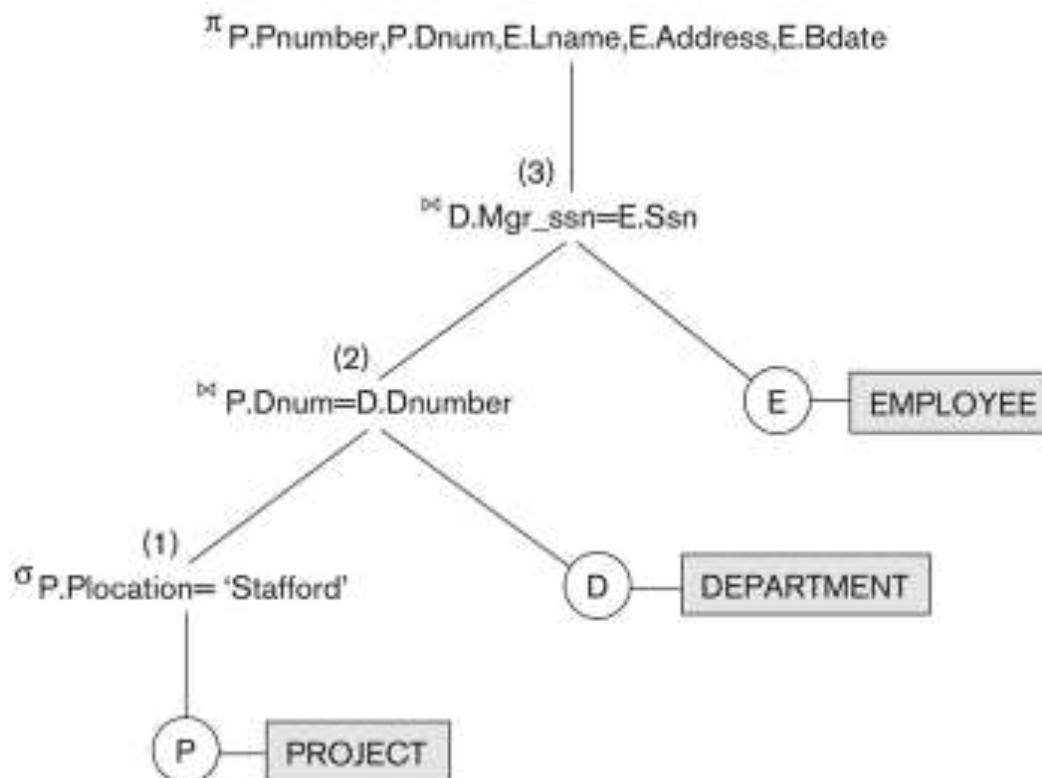
# Query Tree Notation

- **Query Tree**
  - An internal data structure to represent a query
  - Standard technique for estimating the work involved in executing the query, the generation of intermediate results, and the optimization of execution
  - Nodes stand for operations like selection, projection, join, renaming, division, ....
  - Leaf nodes represent base relations
  - A tree gives a good visual feel of the complexity of the query and the operations involved
  - Algebraic Query Optimization consists of rewriting the query or modifying the query tree into an equivalent tree.

**(see Chapter 15)**

# Example of Query Tree

Figure 8.9 Query tree corresponding to the relational algebra expression for Q2.



# 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.

# Aggregate Function Operation

- Use of the Aggregate Functional operation  $\mathcal{F}$ 
  - $\mathcal{F}_{\text{MAX Salary}}$  (EMPLOYEE) retrieves the maximum salary value from the EMPLOYEE relation
  - $\mathcal{F}_{\text{MIN Salary}}$  (EMPLOYEE) retrieves the minimum Salary value from the EMPLOYEE relation
  - $\mathcal{F}_{\text{SUM Salary}}$  (EMPLOYEE) retrieves the sum of the Salary from the EMPLOYEE relation
  - $\mathcal{F}_{\text{COUNT SSN}, \text{AVERAGE Salary}}$  (EMPLOYEE) computes the count (number) of employees and their average salary
    - Note: count just counts the number of rows, without removing duplicates

# Using Grouping with Aggregation

- The previous examples all summarized one or more attributes for a set of tuples
  - Maximum Salary or Count (number of) Ssn
- Grouping can be combined with Aggregate Functions
- Example: For each department, retrieve the DNO, COUNT SSN, and AVERAGE SALARY
- A variation of aggregate operation  $\mathcal{F}$  allows this:
  - Grouping attribute placed to left of symbol
  - Aggregate functions to right of symbol
  - DNO  $\mathcal{F}_{\text{COUNT SSN, AVERAGE Salary}}(\text{EMPLOYEE})$
- Above operation groups employees by DNO (department number) and computes the count of employees and average salary per department

## Figure 8.10 The aggregate function operation.

- a.  $\rho_R(Dno, No\_of\_employees, Average\_sal)(Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE)).$
- b.  $Dno \Sigma alary(EMPLOYEE).$
- c.  $\Sigma COUNT Ssn, AVERAGE Salary(EMPLOYEE).$

R

(a)

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

(b)

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

(c)

| Count_ssn | Average_salary |
|-----------|----------------|
| 8         | 35125          |

# Figure 7.1a Results of GROUP BY and HAVING (in SQL). Q24.

The diagram illustrates the process of generating the result for query Q24 from the EMPLOYEE table. On the left, the EMPLOYEE table is shown with columns: Fname, Minit, Lname, Ssn, ..., Salary, Super\_ssn, and Dno. The rows represent individual employees. A vertical ellipsis between the 5th and 6th columns indicates that many more rows exist, all sharing the same value for Dno (5). An arrow points from this ellipsis to the first row of a summary table on the right. This summary table has three columns: Dno, Count (\*), and Avg (Salary). It contains three rows corresponding to Dno values 5, 4, and 1. The label "Result of Q24" is placed below this summary table.

| Fname    | Minit | Lname   | Ssn       | ... | Salary | Super_ssn | Dno | Dno | Count (*)     | Avg (Salary) |
|----------|-------|---------|-----------|-----|--------|-----------|-----|-----|---------------|--------------|
| John     | B     | Smith   | 123456789 | ... | 30000  | 333445555 | 5   | 5   | 4             | 33250        |
| Franklin | T     | Wong    | 333445555 |     | 40000  | 888665555 | 5   |     | 3             | 31000        |
| Ramesh   | K     | Narayan | 666884444 |     | 38000  | 333445555 | 5   |     | 1             | 55000        |
| Joyce    | A     | English | 453453453 |     | 25000  | 333445555 | 5   | 4   | Result of Q24 |              |
| Alicia   | J     | Zelaya  | 999887777 |     | 25000  | 987654321 | 4   |     |               |              |
| Jennifer | S     | Wallace | 987654321 |     | 43000  | 888665555 | 4   |     |               |              |
| Ahmad    | V     | Jabbar  | 987987987 |     | 25000  | 987654321 | 4   |     |               |              |
| James    | E     | Bong    | 888665555 |     | 55000  | NULL      | 1   |     |               |              |

Grouping EMPLOYEE tuples by the value of Dno

*continued on next slide*

# Additional Relational Operations (continued)

- 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.

# Additional Relational Operations (continued)

- 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.

# Figure 8.11 A two-level recursive query.

| SUPERVISION               |             |
|---------------------------|-------------|
| (Borg's Ssn is 888665555) |             |
| (Ssn)                     | (Super_ssn) |
| 123456789                 | 333445555   |
| 333445555                 | 888665555   |
| 999887777                 | 987654321   |
| 987654321                 | 888665555   |
| 666884444                 | 333445555   |
| 453453453                 | 333445555   |
| 987987987                 | 987654321   |
| 888665555                 | null        |

RESULT1

| Ssn       |
|-----------|
| 333445555 |
| 987654321 |

(Supervised by Borg)

RESULT2

| 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 (continued)

- The OUTER JOIN Operation
  - In NATURAL JOIN and EQUIJOIN, 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.

# Additional Relational Operations (continued)

- The left outer join operation keeps every tuple in the first or left relation R in  $R \bowtie S$ ; if no matching tuple is found in S, then the attributes of S in the join result are filled or “padded” with null values.
- A similar operation, right outer join, keeps every tuple in the second or right relation S in the result of  $R \ltimes S$ .
- A third operation, full outer join, denoted by  $\bowtie\bowtie$  keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with null values as needed.

# Figure 8.12 The result of a LEFT OUTER JOIN operation.

## 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 (continued)

## ■ OUTER UNION Operations

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

# Additional Relational Operations (continued)

- 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 : Procedural Form

- **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.**

$\text{ALL\_EMPS} \leftarrow \pi_{\text{SSN}}(\text{EMPLOYEE})$

$\text{EMPS\_WITH\_DEPS(SSN)} \leftarrow \pi_{\text{ESSN}}(\text{DEPENDENT})$

$\text{EMPS\_WITHOUT\_DEPS} \leftarrow (\text{ALL\_EMPS} - \text{EMPS\_WITH\_DEPS})$

$\text{RESULT} \leftarrow \pi_{\text{LNAME}, \text{FNAME}}(\text{EMPS\_WITHOUT\_DEPS} * \text{EMPLOYEE})$

# Examples of Queries in Relational Algebra – Single expressions

As a single expression, these queries become:

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

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

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

$$\pi_{\text{Lname, Fname}} ((\pi_{\text{Ssn}} (\text{EMPLOYEE}) - \rho_{\text{Ssn}} (\pi_{\text{Essn}} (\text{DEPENDENT}))) * \text{EMPLOYEE})$$

# 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 (continued)

- Relational calculus is considered to be a **nonprocedural** or **declarative** 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  $\text{COND }(t)$  is a conditional expression involving  $t$ .
- The result of such a query is the set of all tuples  $t$  that satisfy  $\text{COND }(t)$ .

# Tuple Relational Calculus (continued)

- 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:

$$\{t.\text{FNAME}, t.\text{LNAME} \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{SALARY} > 50000\}$$

- The condition  $\text{EMPLOYEE}(t)$  specifies that the **range relation** of tuple variable  $t$  is  $\text{EMPLOYEE}$ .
- The first and last name (PROJECTION  $\pi_{\text{FNAME}, \text{LNAME}}$ ) of each  $\text{EMPLOYEE}$  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 are  $(\exists t)(F)$  and  $(\forall t)(F)$ , 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.
  - 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.

# The Existential and Universal Quantifiers (continued)

- $\forall$  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.
- $\exists$  is called the existential or “there exists” quantifier because any tuple that exists in “the universe of” tuples may 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. The query can be expressed as :  
$$\{t.FNAME, t.LNAME, t.ADDRESS \mid EMPLOYEE(t) \text{ and } (\exists d) (DEPARTMENT(d) \text{ and } d.DNAME='Research' \text{ and } d.DNUMBER=t.DNO) \}$$
- 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. The query can be:  
$$\{e.\text{LNAME}, e.\text{FNAME} \mid \text{EMPLOYEE}(e) \text{ and } ((\forall x)(\text{not}(\text{PROJECT}(x)) \text{ or } \text{not}(x.\text{DNUM}=5))$$
  
$$\text{OR } ((\exists w)(\text{WORKS\_ON}(w) \text{ and } w.\text{ESSN}=e.\text{SSN} \text{ and } x.\text{PNUMBER}=w.\text{PNO})))\}$$
- 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.  
$$((\exists w)(\text{WORKS\_ON}(w) \text{ and } w.\text{ESSN}=e.\text{SSN} \text{ and } x.\text{PNUMBER}=w.\text{PNO})$$

# Languages Based on Tuple Relational Calculus

- The language **SQL** is based on tuple calculus. It uses the basic block structure to express the queries in tuple calculus:
  - SELECT <list of attributes>
  - FROM <list of relations>
  - WHERE <conditions>
- 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).

# Languages Based on Tuple Relational Calculus (continued)

- 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. (system is currently still supported by Computer Associates – but the QUEL language is no longer there).

# 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.

# The Domain Relational Calculus (continued)

- 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 :

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

- Abbreviated notation **EMPLOYEE(qrstuvwxyz)** uses the variables without the separating commas: **EMPLOYEE(q,r,s,t,u,v,w,x,y,z)**
- 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, . . . , 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 ( | )—
  - 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 C)

- This language is based on the idea of giving an example of a query using “example elements” which are nothing but domain variables.
- Notation: 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.

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

- 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, and PARADOX.
- For details, see **Appendix C** 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:

# Example Schema as a QBE Query Interface

**EMPLOYEE**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|--------|-----------|-----|
|       |       |       |     |       |         |     |        |           |     |

**DEPARTMENT**

| Dname | Dnumber | Mgr_ssn | Mgr_start_date |
|-------|---------|---------|----------------|
|       |         |         |                |

**DEPT\_LOCATIONS**

| Dnumber | Dlocation |
|---------|-----------|
|         |           |

**PROJECT**

| Pname | Pnumber | Plocation | Dnum |
|-------|---------|-----------|------|
|       |         |           |      |

**WORKS\_ON**

| Essn | Pno | Hours |
|------|-----|-------|
|      |     |       |

**DEPENDENT**

| Essn | Dependent_name | Sex | Bdate | Relationship |
|------|----------------|-----|-------|--------------|
|      |                |     |       |              |

**Figure C.1**

The relational schema of Figure 5.5 as it may be displayed by QBE.

# QBE Examples

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

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

# Four Successive Ways to Specify a QBE Query

(a) EMPLOYEE

| Fname | Minit | Lname | Ssn        | Bdate    | Address                 | Sex | Salary | Super_ssn  | Dno |
|-------|-------|-------|------------|----------|-------------------------|-----|--------|------------|-----|
| John  | B     | Smith | _123456789 | P_9/1/60 | P_100 Main, Houston, TX | _M  | _25000 | _123456789 | _3  |

(b) EMPLOYEE

| Fname | Minit | Lname | Ssn | Bdate    | Address                 | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|----------|-------------------------|-----|--------|-----------|-----|
| John  | B     | Smith |     | P_9/1/60 | P_100 Main, Houston, TX |     |        |           |     |

(c) EMPLOYEE

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|--------|-----------|-----|
| John  | B     | Smith |     | P_X   | P_Y     |     |        |           |     |

(d) EMPLOYEE

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|--------|-----------|-----|
| John  | B     | Smith |     | P.    | P.      |     |        |           |     |

**Figure C.2**

Four ways to specify the query Q0 in QBE.

# 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 C.4(a) gives employees who work on either project 1 or 2, whereas the query in C.4(b) gives those who work on both the projects.

# Complex Conditions with and without a condition box as a part of QBE Query

| WORKS_ON |      |      |
|----------|------|------|
| (a)      | Essn | Pno  |
|          | P.   | > 20 |

| WORKS_ON |      |     |
|----------|------|-----|
| (b)      | Essn | Pno |
|          | P.   | _PX |

## CONDITIONS

\_HX > 20 and (PX = 1 or PX = 2)

| WORKS_ON |      |     |
|----------|------|-----|
| (c)      | Essn | Pno |
|          | P.   | 1   |
|          | P.   | 2   |

**Figure C.3**

Specifying complex conditions in QBE. (a) The query Q0A. (b) The query Q0B with a condition box. (c) The query Q0B without a condition box.

# Handling AND conditions in a QBE Query

**WORKS\_ON**

(a)

| Essn | Pno | Hours |
|------|-----|-------|
| P_ES | 1   |       |
| P_ES | 2   |       |

**WORKS\_ON**

(b)

| Essn | Pno | Hours |
|------|-----|-------|
| P_EX | 1   |       |
| P_EY | 2   |       |

**CONDITIONS**

$_EX = _EY$

**Figure C.4**

Specifying EMPLOYEES who work on both projects. (a) Incorrect specification of an AND condition. (b) Correct specification.

# JOIN in QBE : Examples

- The join is simply accomplished by using the same example element (variable with underscore) in the columns being joined from different (or same as in C.5 (b)) relation.
- Note that the Result is set us as an independent table to show variables from multiple relations placed in the result.

# Performing Join with common example elements and use of a RESULT relation

**Figure C.5**

Illustrating JOIN and result relations in QBE. (a) The query Q1. (b) The query Q8.

**(a) EMPLOYEE**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|--------|-----------|-----|
| _FN   |       | _LN   |     |       | _Addr   |     |        |           | _DX |

**DEPARTMENT**

| Dname    | Dnumber | Mgrssn | Mgr_start_date |
|----------|---------|--------|----------------|
| Research | _DX     |        |                |

| RESULT |     |     |       |
|--------|-----|-----|-------|
| P.     | _FN | _LN | _Addr |

**(b) EMPLOYEE**

| Fname | Minit | Lname | Ssn   | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-------|-------|---------|-----|--------|-----------|-----|
| _E1   |       | _E2   |       |       |         |     |        | _Xssn     |     |
| _S1   |       | _S2   | _Xssn |       |         |     |        |           |     |

| RESULT |     |     |     |     |
|--------|-----|-----|-----|-----|
| P.     | _E1 | _E2 | _S1 | _S2 |

# AGGREGATION in QBE

- Aggregation is accomplished by using .CNT for count,.MAX, .MIN, .AVG for the corresponding aggregation functions
- Grouping is accomplished by .G operator.
- Condition Box may use conditions on groups (similar to HAVING clause in SQL – see Section 8.5.8)

# AGGREGATION in QBE : Examples

(a) EMPLOYEE

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|--------|-----------|-----|
|       |       |       |     |       |         |     | P.CNT. |           |     |

(b) EMPLOYEE

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary    | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|-----------|-----------|-----|
|       |       |       |     |       |         |     | P.CNT.ALL |           |     |

(c) EMPLOYEE

| Fname | Minit | Lname | Ssn       | Bdate | Address | Sex | Salary    | Super_ssn | Dno  |
|-------|-------|-------|-----------|-------|---------|-----|-----------|-----------|------|
|       |       |       | P.CNT.ALL |       |         |     | P.AVG.ALL |           | P.G. |

(d) PROJECT

| Pname | Pnumber | Plocation | Dnum |
|-------|---------|-----------|------|
| P.    | _PX     |           |      |

WORKS\_ON

| Essn     | Pno   | Hours |
|----------|-------|-------|
| P.CNT.EX | G._PX |       |

CONDITIONS

|            |
|------------|
| CNT_EX > 2 |
|------------|

**Figure C.6**

Functions and grouping in QBE. (a) The query Q23.  
(b) The query Q23A. (c) The query Q24. (d) The query Q26.

# NEGATION in QBE : Example

**Figure C.7**

Illustrating negation by the query Q6.

## EMPLOYEE

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|-------|-------|-------|-----|-------|---------|-----|--------|-----------|-----|
| P.    |       | P.    | _SX |       |         |     |        |           |     |

## DEPENDENT

| Essn | Dependent_name | Sex | Bdate | Relationship |
|------|----------------|-----|-------|--------------|
| _SX  |                |     |       |              |

# UPDATING in QBE : Examples

(a)

## EMPLOYEE

| I. | Fname   | Minit | Lname  | Ssn       | Bdate     | Address                 | Sex | Salary | Super_ssn | Dno |
|----|---------|-------|--------|-----------|-----------|-------------------------|-----|--------|-----------|-----|
| I. | Richard | K     | Marini | 653298653 | 30-Dec-52 | 98 Oak Forest, Katy, TX | M   | 37000  | 987654321 | 4   |

(b)

## EMPLOYEE

| D. | Fname | Minit | Lname | Ssn       | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|----|-------|-------|-------|-----------|-------|---------|-----|--------|-----------|-----|
| D. |       |       |       | 653298653 |       |         |     |        |           |     |

(c)

## EMPLOYEE

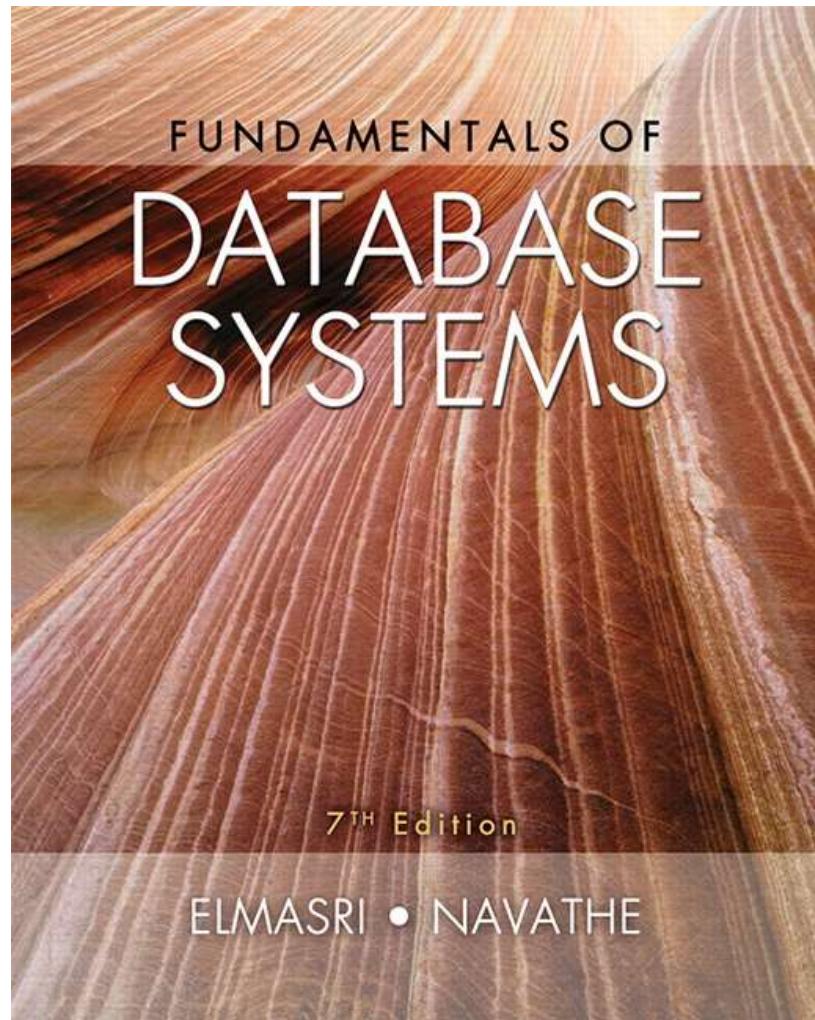
|  | Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary   | Super_ssn | Dno |
|--|-------|-------|-------|-----|-------|---------|-----|----------|-----------|-----|
|  | John  |       | Smith |     |       |         |     | U._S*1.1 |           | U.4 |

**Figure C.8**

Modifying the database in QBE. (a) Insertion. (b) Deletion. (c) Update in QBE.

# Chapter Summary

- 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 C)



# CHAPTER 9

## 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).

# GOALS during Mapping

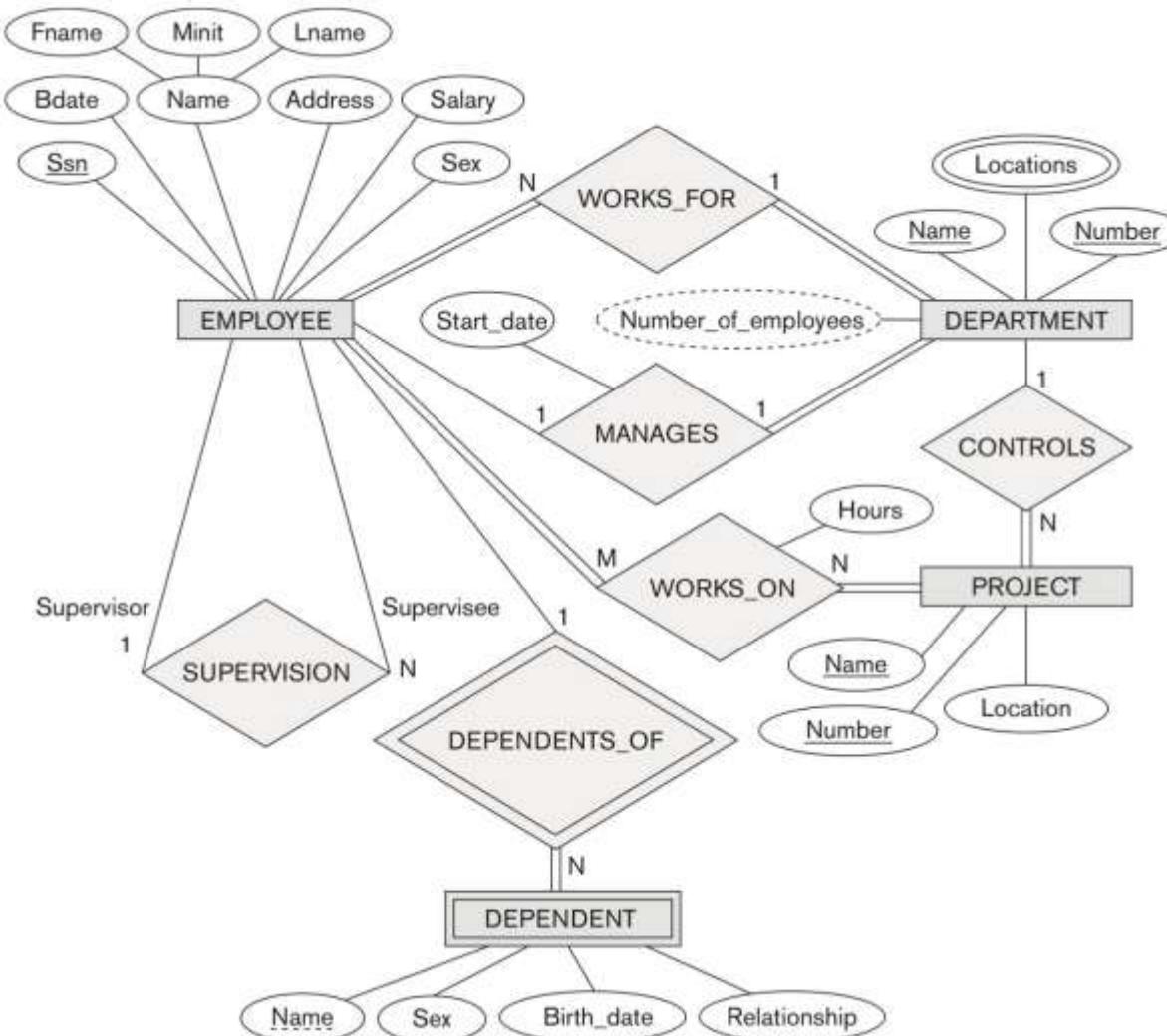
- Preserve all information (that includes all attributes)
- Maintain the constraints to the extent possible (Relational Model cannot preserve all constraints- e.g., max cardinality ratio such as 1:10 in ER; exhaustive classification into subtypes, e.g., STUDENTS are specialized into Domestic and Foreign)
- Minimize null values

*The mapping procedure described has been implemented in many commercial tools.*

# 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 9.1 The ER conceptual schema diagram for the COMPANY database.



# ER-to-Relational Mapping Algorithm (contd.)

## ■ 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 & include all simple attributes (or simple components of composite attributes) of W as attributes of R.
- Also, 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 (contd.)

- **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 ( 2 relations) approach:** Choose one of the relations-say S-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 (1 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 ( 3 relations) 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 (contd.)

- 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.
- An alternative approach is to use a Relationship relation (cross referencing relation) – this is rarely done.

# ER-to-Relational Mapping Algorithm (contd.)

- **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. This is a relationship relation.*
  - 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 (contd.)

## ■ 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 or 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.

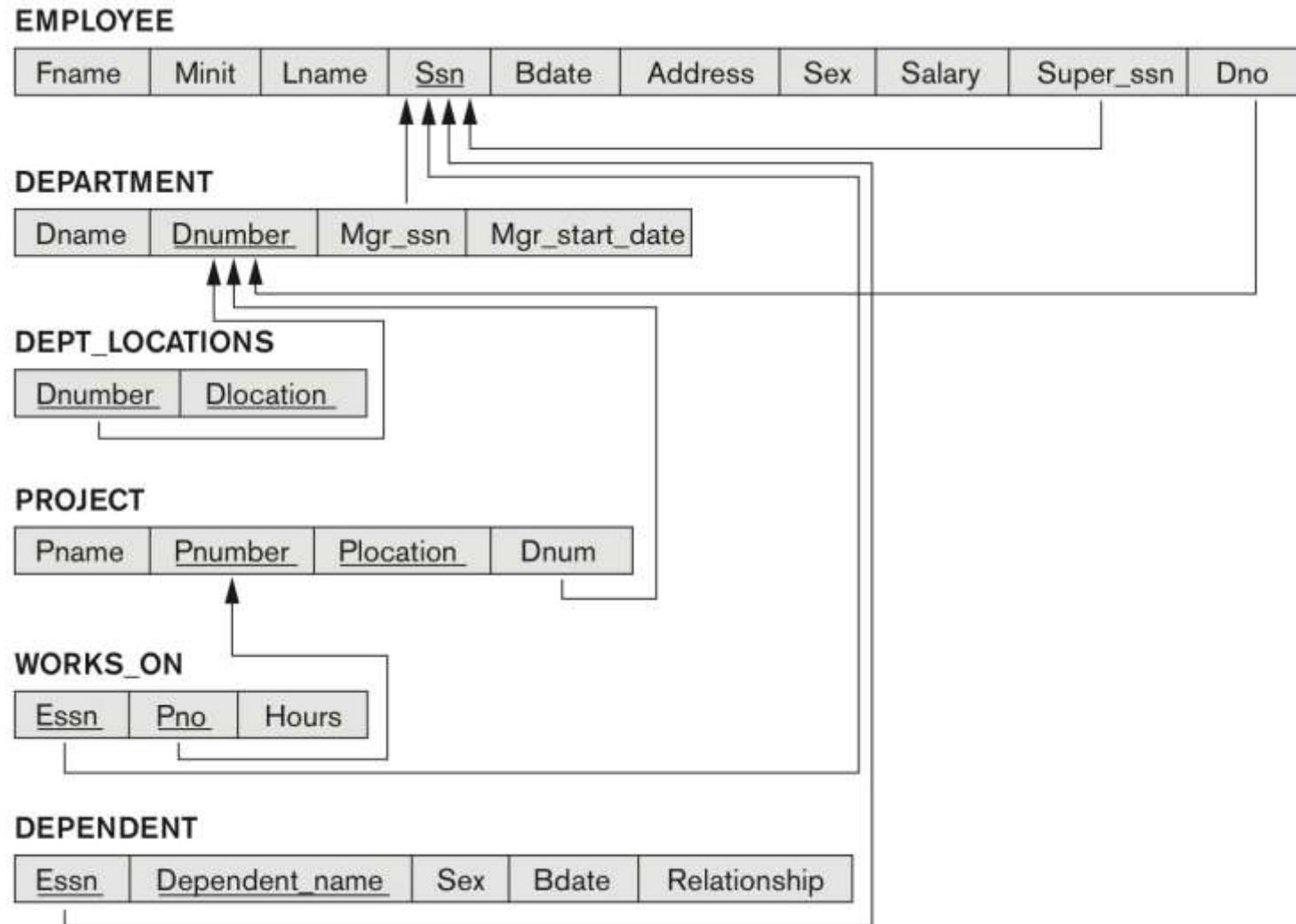
## ■ Example: The relation DEPT\_LOCATIONS is created.

- 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}.

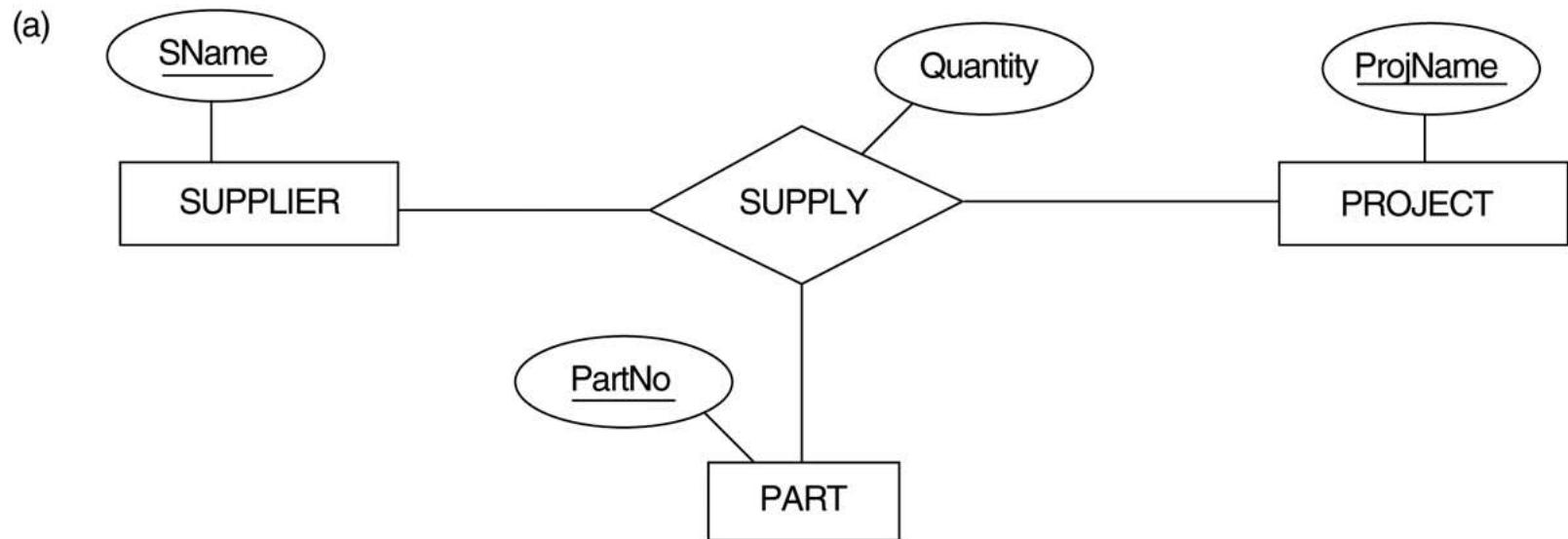
# ER-to-Relational Mapping Algorithm (contd.)

- **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 on the next slide.
  - 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}

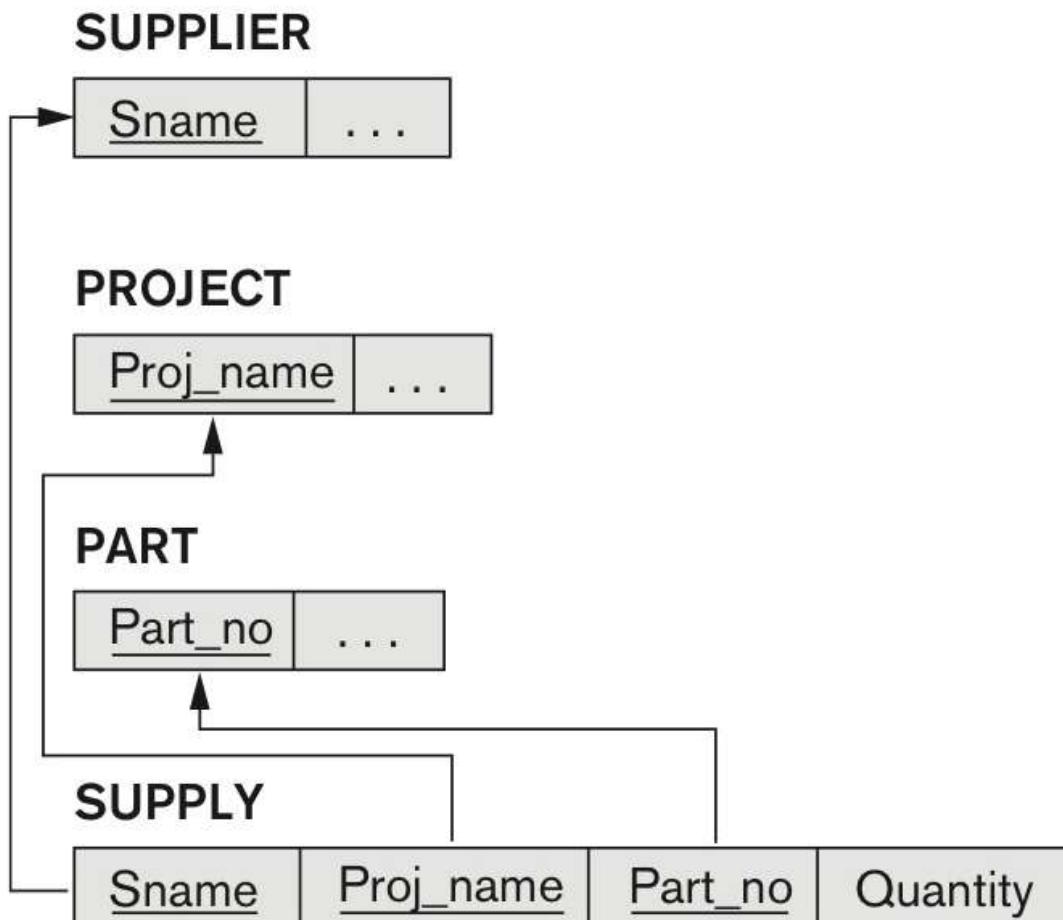
# Figure 9.2 Result of mapping the COMPANY ER schema into a relational database schema.



## FIGURE 3.17 TERNARY RELATIONSHIP: SUPPLY



# Mapping the *n*-ary relationship type SUPPLY



# Summary of Mapping constructs and constraints

**Table 9.1** Correspondence between ER and Relational Models

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

# Mapping of Generalization and Specialization Hierarchies to a Relational Schema

# Mapping EER Model Constructs to Relations

- **Step8: Options for Mapping Specialization or Generalization.**
  - Convert each specialization with m subclasses {S1, S2, ..., Sm} and generalized superclass C, where the attributes of C are {k, a1, ... an} and k is the (primary) key, into relational schemas using one of the four following options:
    - Option 8A: Multiple relations-Superclass and subclasses
    - Option 8B: Multiple relations-Subclass relations only
    - Option 8C: Single relation with one type attribute
    - Option 8D: Single relation with multiple type attributes

# Mapping EER Model Constructs to Relations

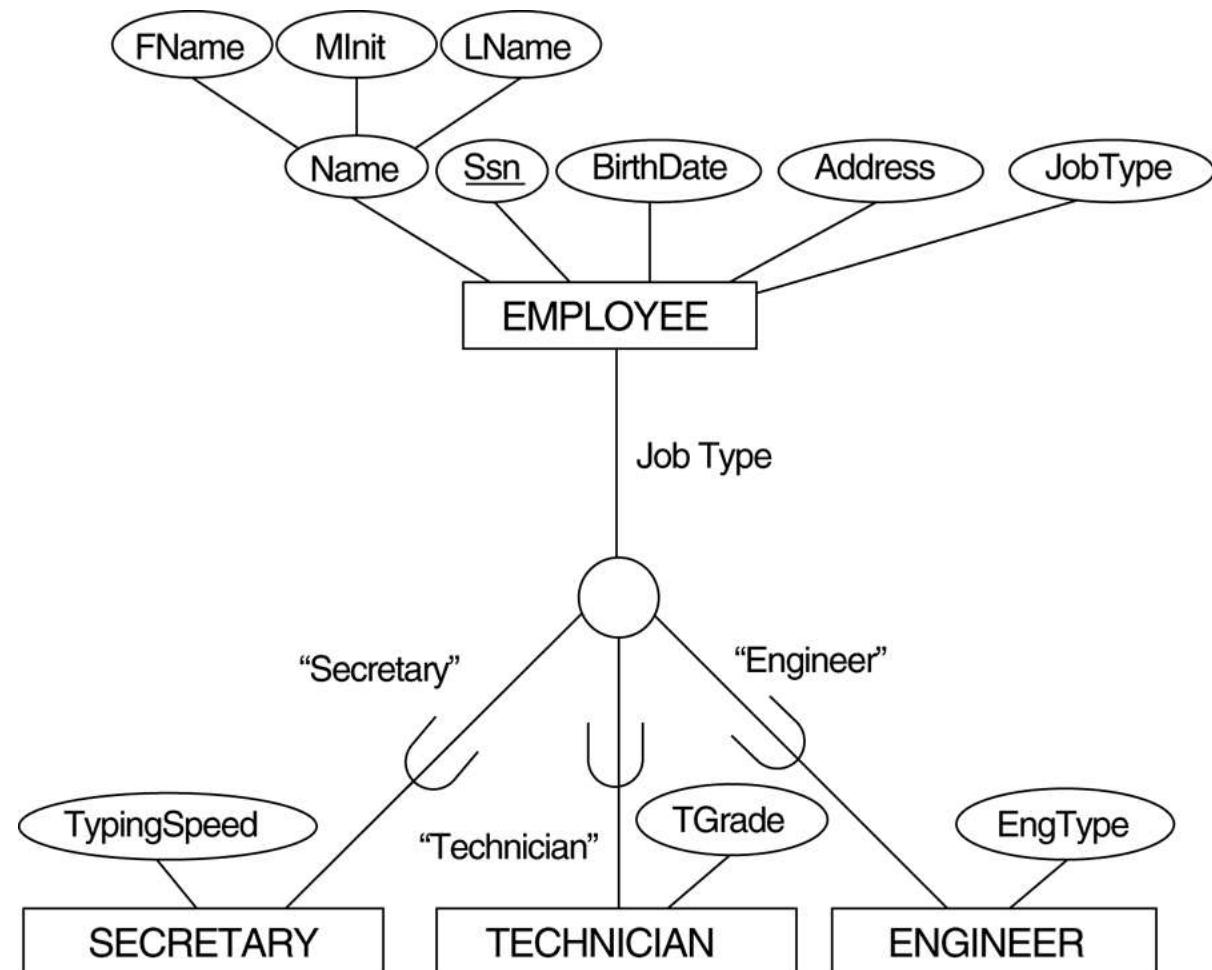
- **Option 8A: Multiple relations-Superclass and subclasses**
  - Create a relation L for C with attributes  $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$  and  $\text{PK}(L) = k$ . Create a relation  $L_i$  for each subclass  $S_i$ ,  $1 < i < m$ , with the attributes  $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$  and  $\text{PK}(L_i) = k$ . This option works for any specialization (total or partial, disjoint or over-lapping).
- **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).

# Mapping EER Model Constructs to Relations (contd.)

- **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.



# 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 |
|------------|---------|

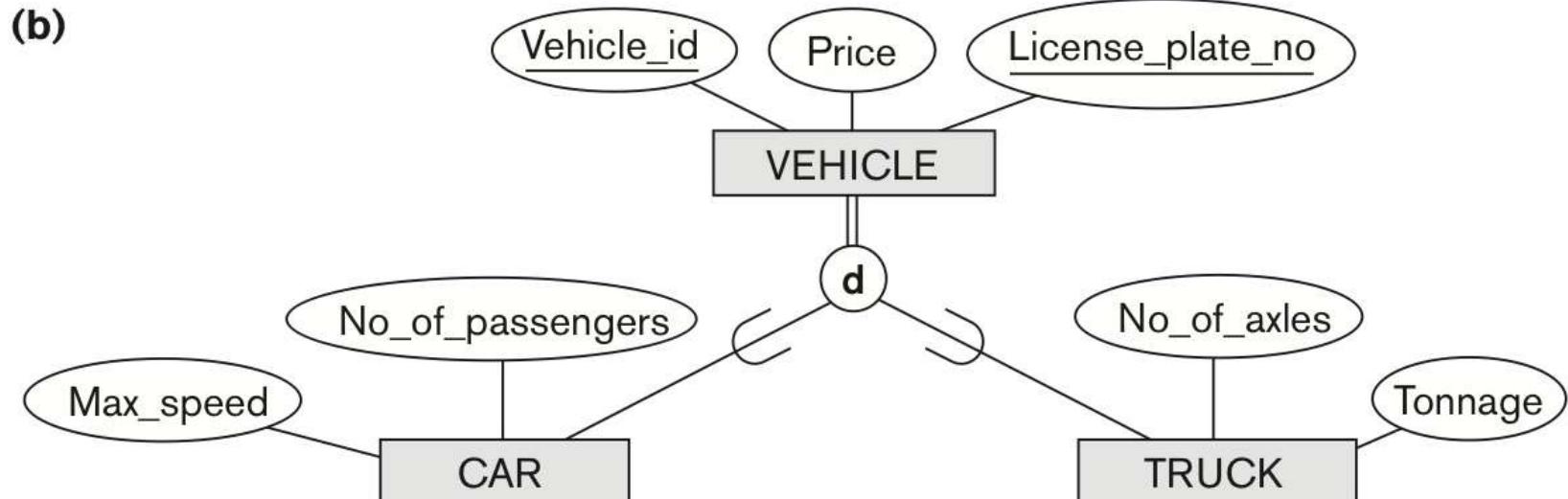
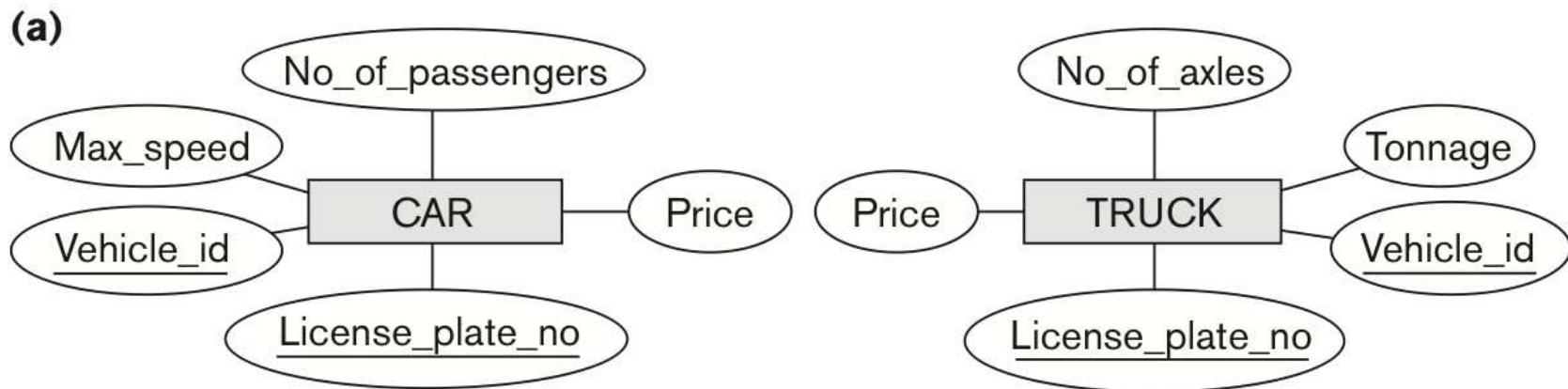
# Mapping the EER schema in Figure 4.4 using option 8C

(c) EMPLOYEE

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

## FIGURE 4.3 (b)

Generalizing CAR and TRUCK into the superclass VEHICLE.



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

(b) CAR

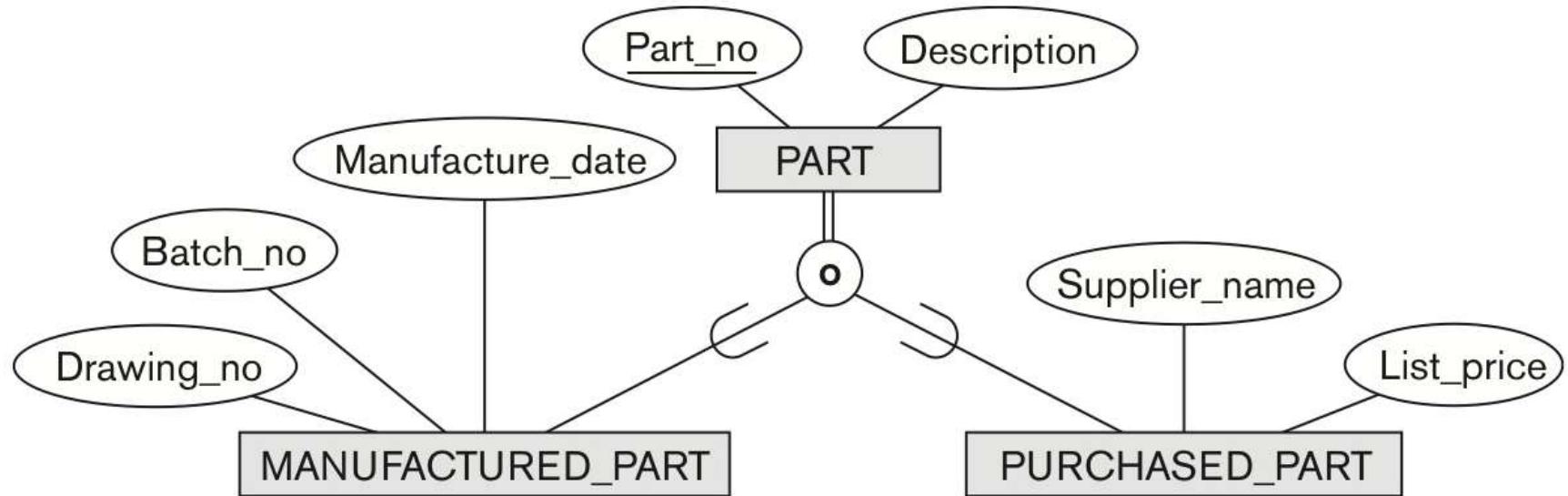
|                  |                |       |          |                |
|------------------|----------------|-------|----------|----------------|
| <u>VehicleId</u> | LicensePlateNo | Price | MaxSpeed | NoOfPassengers |
|------------------|----------------|-------|----------|----------------|

TRUCK

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

## FIGURE 4.5

An overlapping (non-disjoint) specialization.



# Mapping Figure 4.5 using option 8D with Boolean type fields Mflag and Pflag.

(d) PART

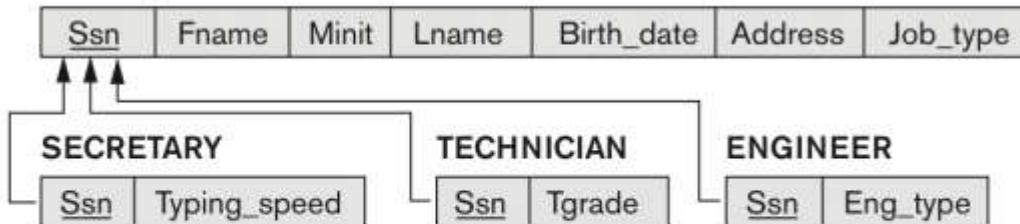
| PartNo | Description | MFlag | DrawingNo | ManufactureDate | BatchNo | PFlag | SupplierName | ListPrice |
|--------|-------------|-------|-----------|-----------------|---------|-------|--------------|-----------|
|--------|-------------|-------|-----------|-----------------|---------|-------|--------------|-----------|

# Different Options for Mapping Generalization Hierarchies

- **Next Slide :Figure 9.5** Options for mapping specialization or generalization.
  - (a) Mapping the EER schema in Figure 4.4 using option 8A.
  - (b) Mapping the EER schema in Figure 4.3(b) using option 8B.
  - (c) Mapping the EER schema in Figure 4.4 using option 8C.
  - (d) Mapping Figure 4.5 using option 8D with Boolean type fields Mflag and Pflag.

# Fig. 9.5: Different Options for Mapping Generalization Hierarchies - summary

## (a) EMPLOYEE



## (b) CAR

|            |                  |       |           |                  |
|------------|------------------|-------|-----------|------------------|
| Vehicle_id | License_plate_no | Price | Max_speed | No_of_passengers |
|------------|------------------|-------|-----------|------------------|

## TRUCK

|            |                  |       |             |         |
|------------|------------------|-------|-------------|---------|
| Vehicle_id | License_plate_no | Price | No_of_axles | Tonnage |
|------------|------------------|-------|-------------|---------|

## (c) EMPLOYEE

|     |       |       |       |            |         |          |              |        |          |
|-----|-------|-------|-------|------------|---------|----------|--------------|--------|----------|
| Ssn | Fname | Minit | Lname | Birth_date | Address | Job_type | Typing_speed | Tgrade | Eng_type |
|-----|-------|-------|-------|------------|---------|----------|--------------|--------|----------|

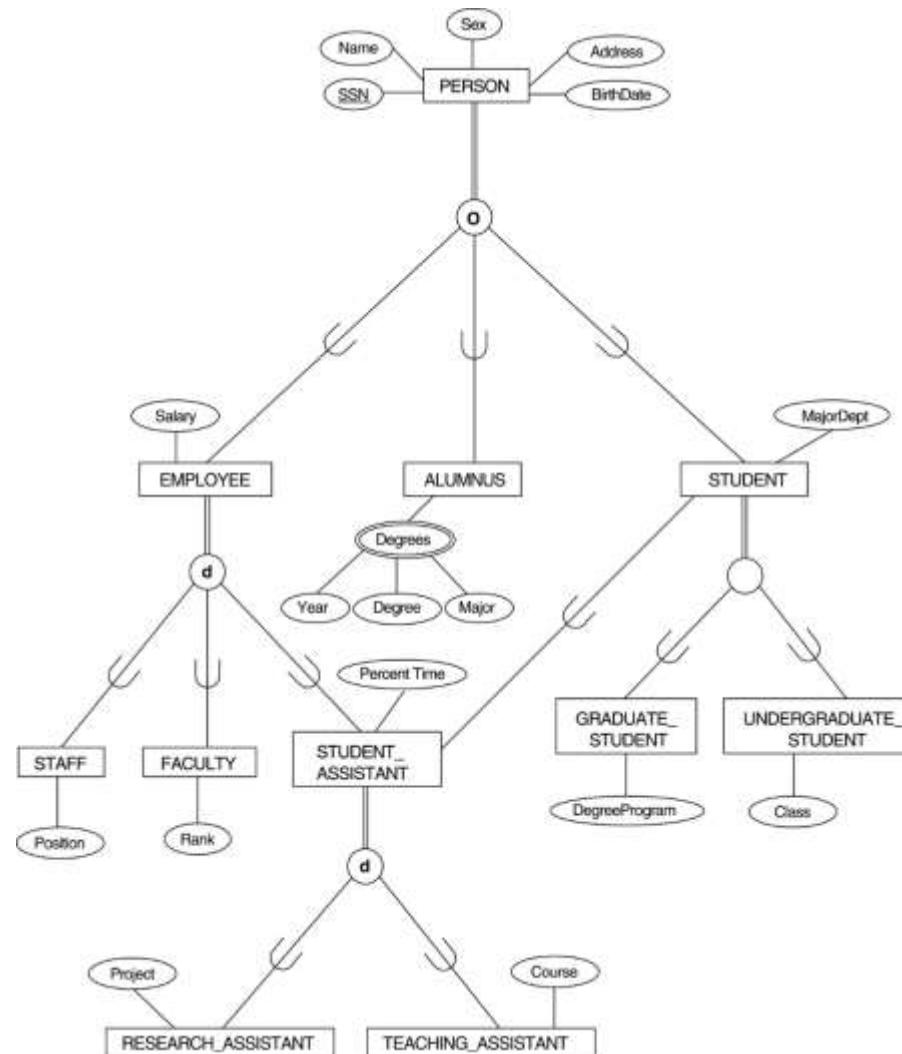
## (d) PART

|         |             |       |            |                  |          |       |               |            |
|---------|-------------|-------|------------|------------------|----------|-------|---------------|------------|
| Part_no | Description | Mflag | Drawing_no | Manufacture_date | Batch_no | Pflag | Supplier_name | List_price |
|---------|-------------|-------|------------|------------------|----------|-------|---------------|------------|

# Mapping EER Model Constructs to Relations (contd.)

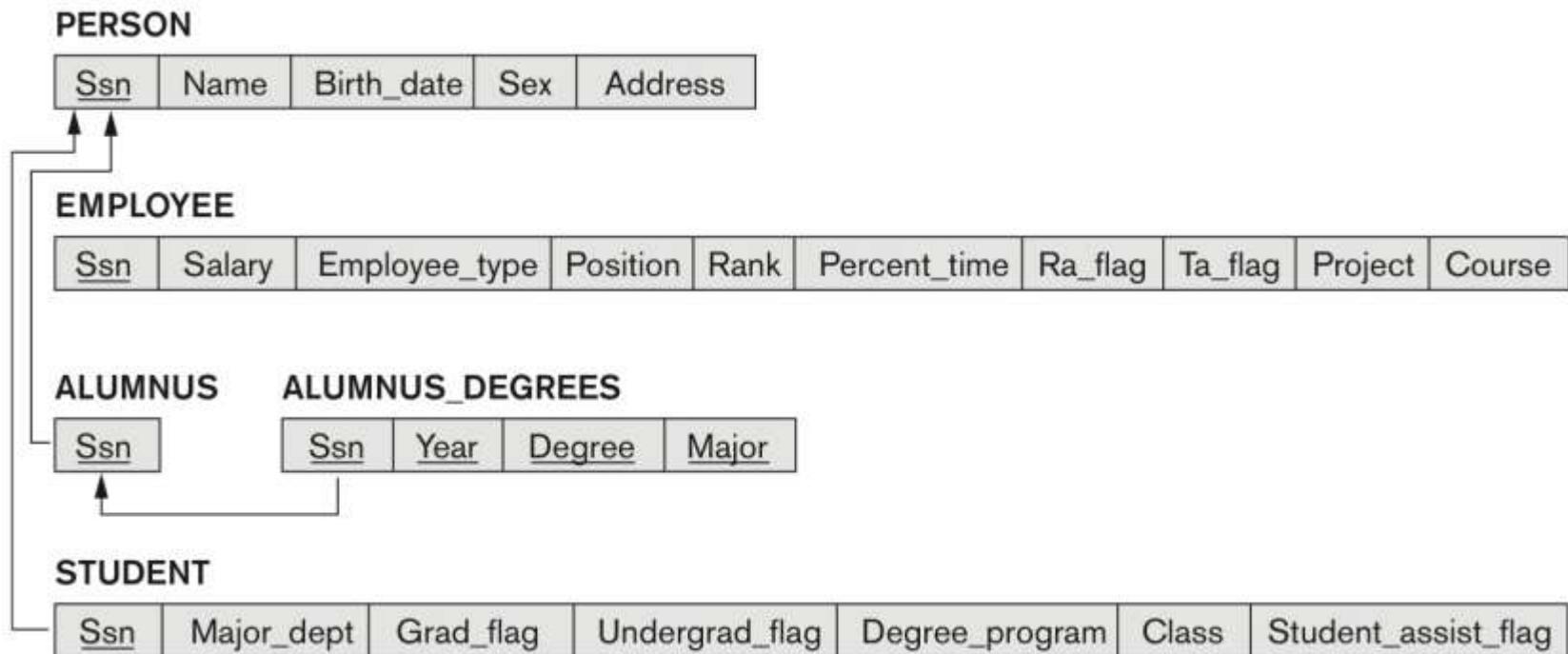
- Mapping of Shared Subclasses (Multiple Inheritance)
  - A shared subclass, such as STUDENT\_ASSISTANT, is a subclass of several classes, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category.
  - We can apply 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 9.6

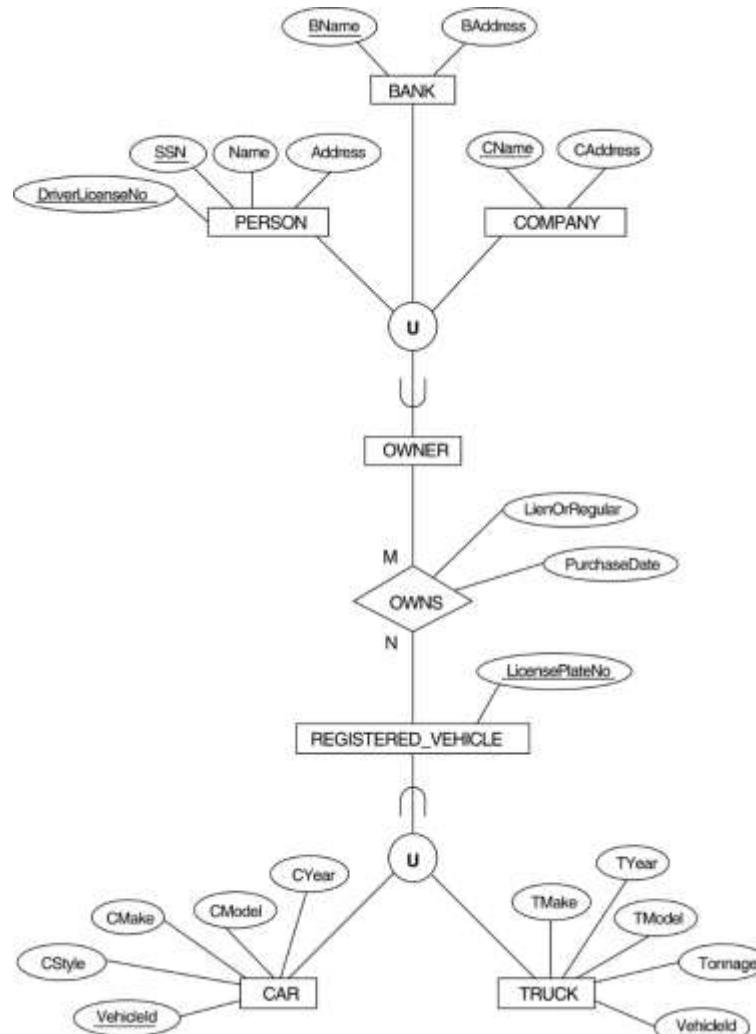
Mapping the EER specialization lattice in Figure 4.7 using multiple options.



# Mapping EER Model Constructs to Relations (contd.)

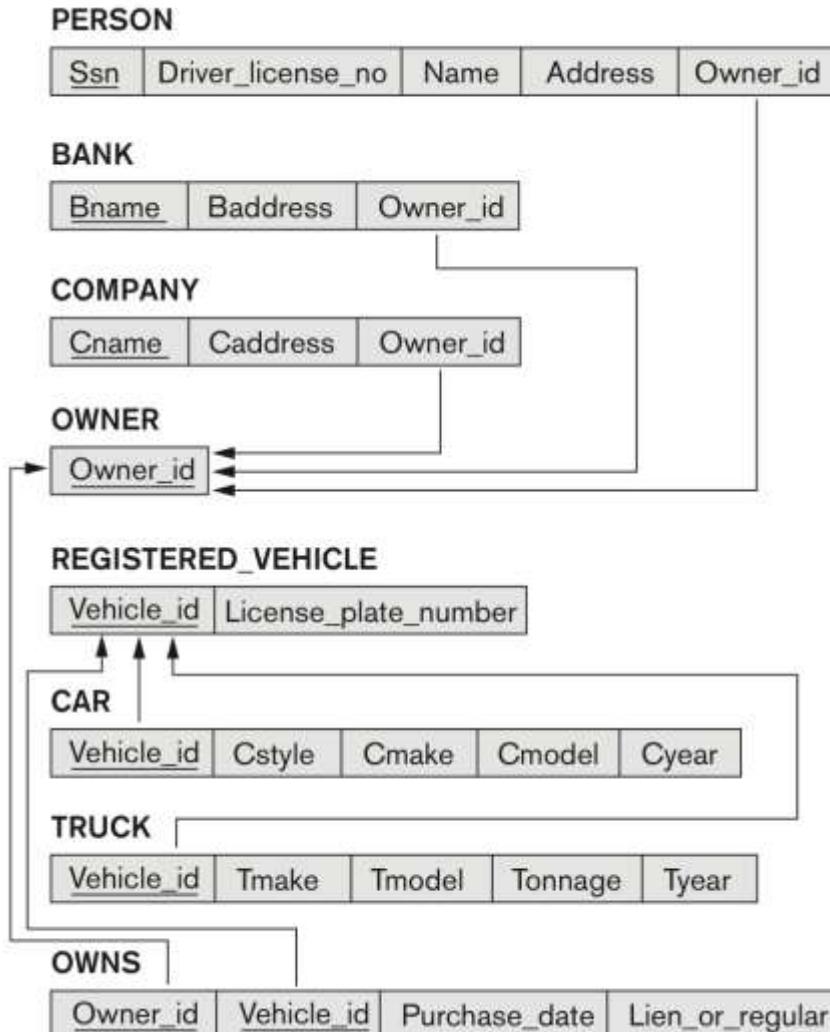
- **Step 9: Mapping of Union Types (Categories).**
  - For mapping a category whose defining superclass have different keys, it is customary to specify a new key attribute, called a surrogate key, when creating a relation to correspond to the category.
  - In the example below we can create a relation 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.



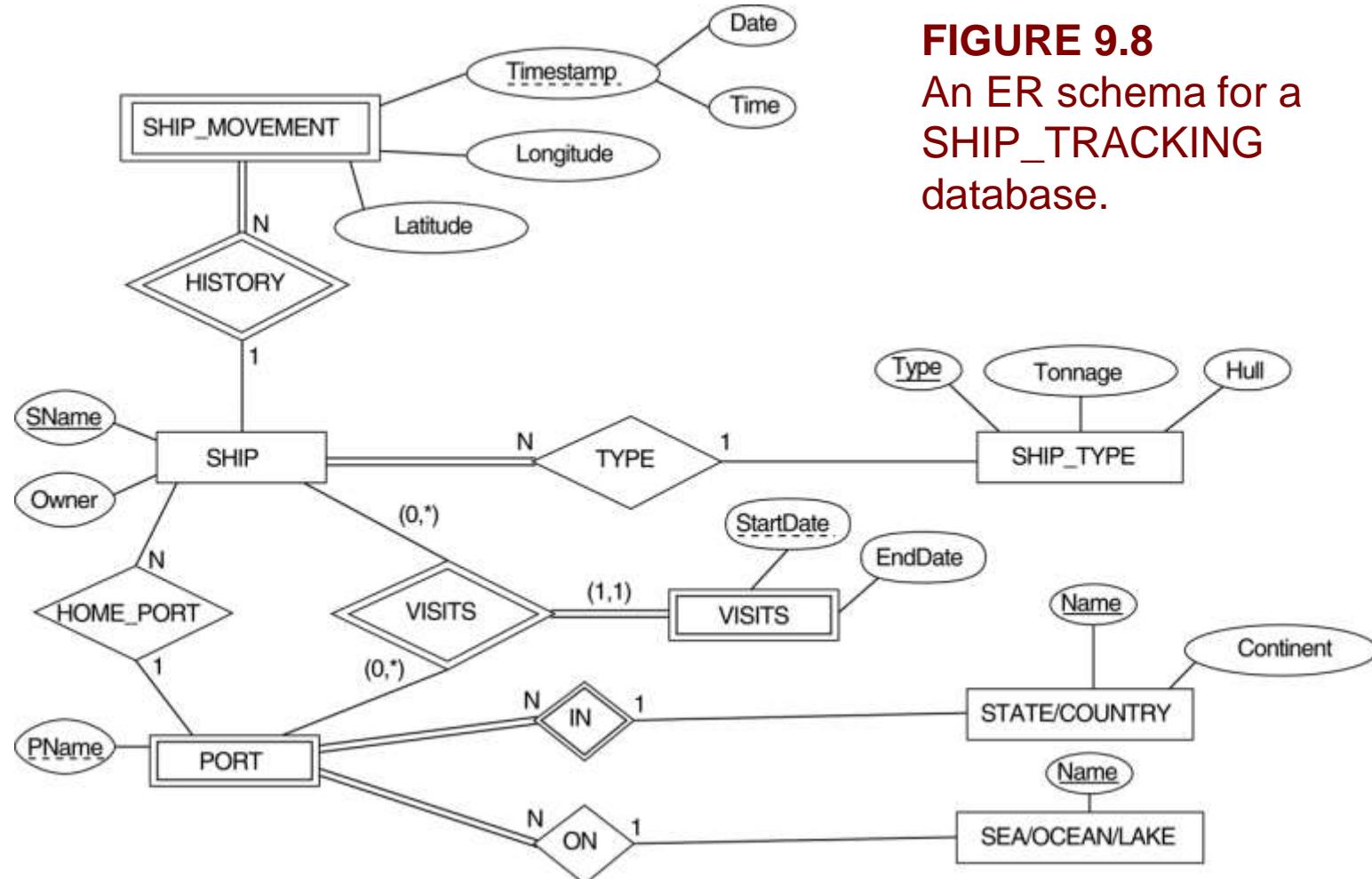
## FIGURE 9.7

Mapping the EER categories (union types) in Figure 4.8 to relations.



# Mapping Exercise-1

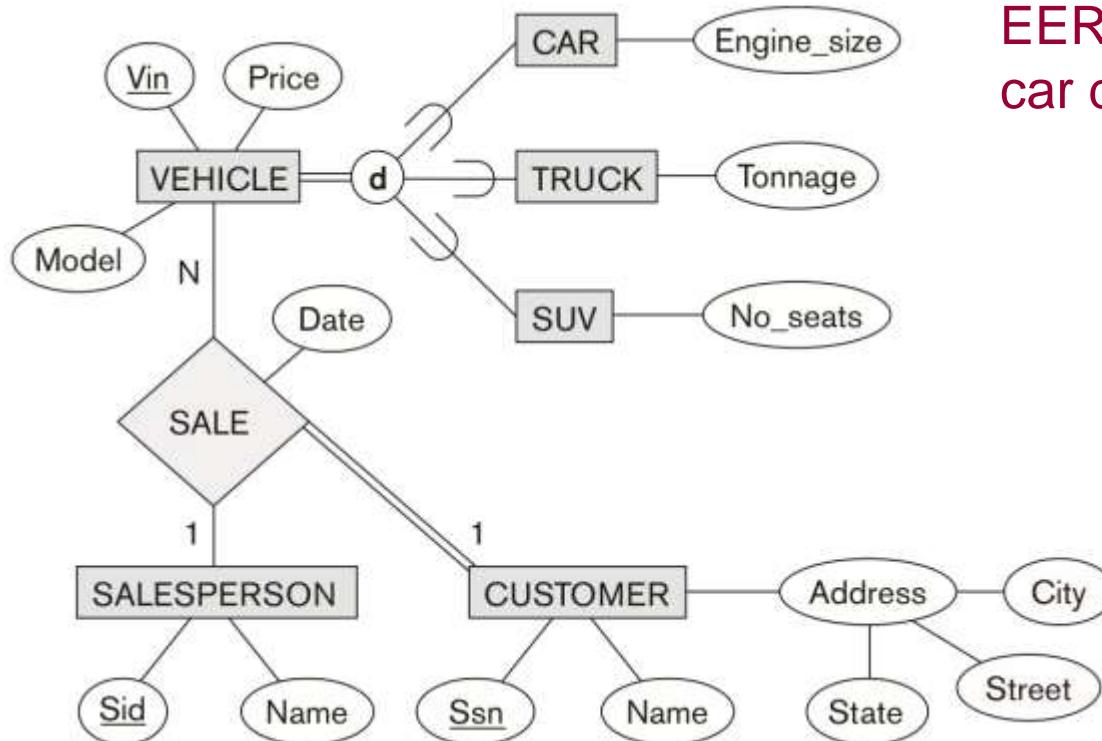
Exercise 9.4 : Map this schema into a set of relations.



**FIGURE 9.8**  
An ER schema for a  
SHIP\_TRACKING  
database.

# Mapping Exercise-2

Exercise 9.9 : Map this schema into a set of relations



**FIGURE 9.9**  
EER diagram for a  
car dealer

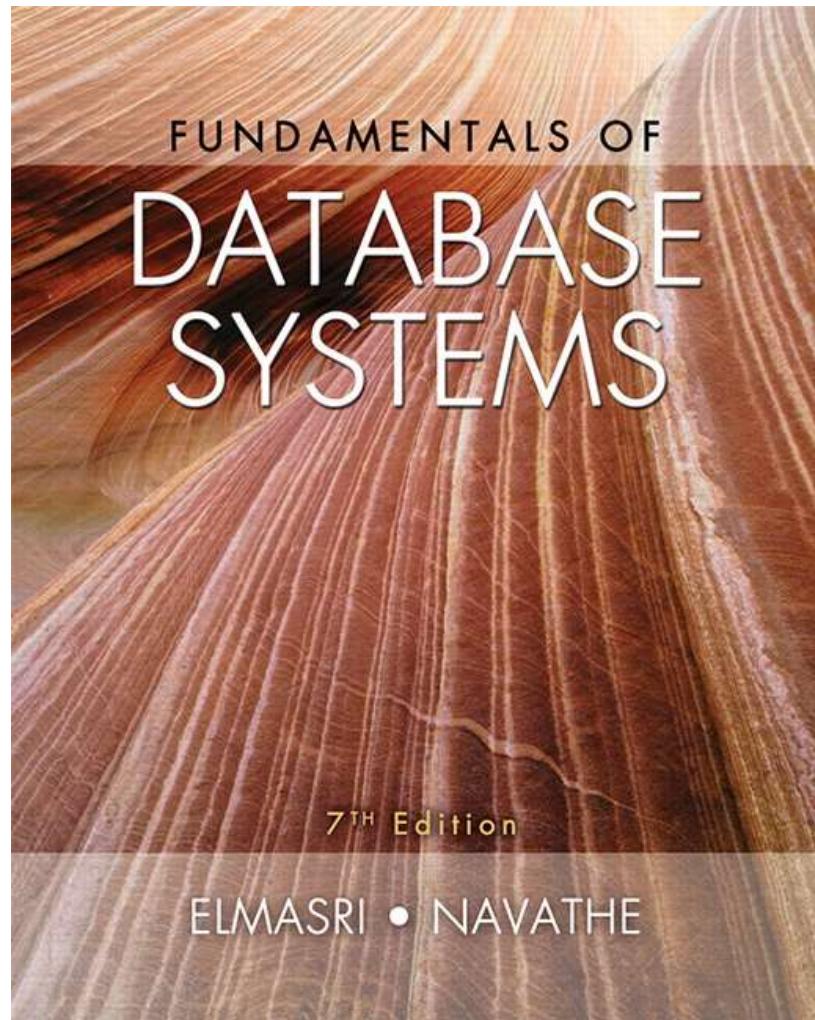
# Chapter Summary

- **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).



# Chapter 10 Outline

- Database Programming: Techniques and Issues
- Embedded SQL, Dynamic SQL, and SQLJ
- Database Programming with Function Calls:  
SQL/CLI and JDBC
- Database Stored Procedures  
and SQL/PSM
- Comparing the Three Approaches

# Introduction to SQL Programming Techniques

- **Database applications**
  - Host language
    - Java, C/C++/C#, COBOL, or some other programming language
  - Data sublanguage
    - SQL
- **SQL standards**
  - Continually evolving
  - Each DBMS vendor may have some variations from standard

# Database Programming: Techniques and Issues

- **Interactive interface**
  - SQL commands typed directly into a monitor
- **Execute file of commands**
  - `@<filename>`
- **Application programs or database applications**
  - Used as canned transactions by the end users access a database
  - May have **Web interface**

# Approaches to Database Programming

- **Embedding** database commands in a general-purpose programming language
  - Database statements identified by a special prefix
  - **Precompiler or preprocessor** scans the source program code
    - Identify database statements and extract them for processing by the DBMS
  - Called **embedded SQL**

# Approaches to Database Programming (cont'd.)

- Using a library of database functions
  - **Library of functions** available to the host programming language
  - **Application programming interface (API)**
- Designing a brand-new language
  - **Database programming language** designed from scratch
- First two approaches are more common

# Impedance Mismatch

- Differences between database model and programming language model
- **Binding** for each host programming language
  - Specifies for each attribute type the compatible programming language types
- Cursor or iterator variable
  - Loop over the tuples in a query result

# Typical Sequence of Interaction in Database Programming

- Open a connection to database server
- Interact with database by submitting queries, updates, and other database commands
- Terminate or close connection to database

# Embedded SQL, Dynamic SQL, and SQLJ

- **Embedded SQL**
  - C language
- **SQLJ**
  - Java language
- Programming language called **host language**

# Retrieving Single Tuples with Embedded SQL

- EXEC SQL
  - Prefix
  - **Preprocessor** separates embedded SQL statements from host language code
  - Terminated by a matching END-EXEC
    - Or by a semicolon (;
- **Shared variables**
  - Used in both the C program and the embedded SQL statements
  - Prefixed by a colon (:) in SQL statement

**Figure 10.1** C program variables used in the embedded SQL examples E1 and E2.

```
0) int loop ;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar dname [16], fname [16], lname [16], address [31] ;
3) char ssn [10], bdate [11], sex [2], minit [2] ;
4) float salary, raise ;
5) int dno, dnumber ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

# Retrieving Single Tuples with Embedded SQL (cont'd.)

- Connecting to the database

```
CONNECT TO <server name>AS <connection name>
AUTHORIZATION <user account name and password> ;
```

- Change connection

```
SET CONNECTION <connection name> ;
```

- Terminate connection

```
DISCONNECT <connection name> ;
```

# Retrieving Single Tuples with Embedded SQL (cont'd.)

- **SQLCODE** and **SQLSTATE** communication variables
  - Used by DBMS to communicate exception or error conditions
- **SQLCODE** variable
  - 0 = statement executed successfully
  - 100 = no more data available in query result
  - < 0 = indicates some error has occurred

# Retrieving Single Tuples with Embedded SQL (cont'd.)

## ■ SQLSTATE

- String of five characters
- '00000' = no error or exception
- Other values indicate various errors or exceptions
- For example, '02000' indicates 'no more data' when using SQLSTATE

## **Figure 10.2** Program segment E1, a C program segment with embedded SQL.

```
//Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2) prompt("Enter a Social Security Number: ", ssn) ;
3) EXEC SQL
4) SELECT Fname, Minit, Lname, Address, Salary
5) INTO :fname, :minit, :lname, :address, :salary
6) FROM EMPLOYEE WHERE Ssn = :ssn ;
7) if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8) else printf("Social Security Number does not exist: ", ssn) ;
9) prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10) }
```

# Retrieving Multiple Tuples with Embedded SQL Using Cursors

- Cursor
  - Points to a single tuple (row) from result of query
- **OPEN CURSOR** command
  - Fetches query result and sets cursor to a position before first row in result
  - Becomes current row for cursor
- **FETCH** commands
  - Moves cursor to next row in result of query

**Figure 10.3** Program segment E2, a C program segment that uses cursors with embedded SQL

```
//Program Segment E2:
0) prompt("Enter the Department Name: ", dname) ;
1) EXEC SQL
2) SELECT Dnumber INTO :dnumber
3) FROM DEPARTMENT WHERE Dname = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5) SELECT Ssn, Fname, Minit, Lname, Salary
6) FROM EMPLOYEE WHERE Dno = :dnumber
7) FOR UPDATE OF Salary ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11) printf("Employee name is:", Fname, Minit, Lname) ;
12) prompt("Enter the raise amount: ", raise) ;
13) EXEC SQL
14) UPDATE EMPLOYEE
15) SET Salary = Salary + :raise
16) WHERE CURRENT OF EMP ;
17) EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
18) }
19) EXEC SQL CLOSE EMP ;
```

# Retrieving Multiple Tuples with Embedded SQL Using Cursors (cont'd.)

- **FOR UPDATE OF**
  - List the names of any attributes that will be updated by the program
- Fetch orientation
  - Added using value: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *i*, and RELATIVE *i*

```
DECLARE <cursor name> [INSENSITIVE] [SCROLL] CURSOR
[WITH HOLD] FOR <query specification>
[ORDER BY <ordering specification>]
[FOR READ ONLY | FOR UPDATE [OF <attribute list>]] ;
```

# Specifying Queries at Runtime Using Dynamic SQL

- **Dynamic SQL**
  - Execute different SQL queries or updates dynamically at runtime
- Dynamic update
- Dynamic query

**Figure 10.4** Program segment E3, a C program segment that uses dynamic SQL for updating a table.

```
//Program Segment E3:
0) EXEC SQL BEGIN DECLARE SECTION ;
1) varchar sqlupdatestring [256] ;
2) EXEC SQL END DECLARE SECTION ;
 ...
3) prompt("Enter the Update Command: ", sqlupdatestring) ;
4) EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5) EXEC SQL EXECUTE sqlcommand ;
 ...
```

# SQLJ: Embedding SQL Commands in Java

- Standard adopted by several vendors for embedding SQL in Java
- Import several class libraries
- **Default context**
- Uses **exceptions** for error handling
  - `SQLException` is used to return errors or exception conditions

**Figure 10.5** Importing classes needed for including SQLJ in Java programs in Oracle, and establishing a connection and default context.

```
1) import java.sql.* ;
2) import java.io.* ;
3) import sqlj.runtime.*;
4) import sqlj.runtime.ref.*;
5) import oracle.sqlj.runtime.*;

...
6) DefaultContext ctxt =
7) oracle.getConnection("<url name>", "<user name>", "<password>", true) ;
8) DefaultContext.setDefaultContext(ctxt) ;

...
```

**Figure 10.6** Java program variables used in SQLJ examples J1 and J2.

```
1) string dname, ssn , fname, fn, lname, ln,
 bdate, address ;
2) char sex, minit, mi ;
3) double salary, sal ;
4) integer dno, dnumber ;
```

**Figure 10.7** Program segment J1, a Java program segment with SQLJ.

```
//Program Segment J1:
1) ssn = readEntry("Enter a Social Security Number: ") ;
2) try {
3) #sql { SELECT Fname, Minit, Lname, Address, Salary
4) INTO :fname, :minit, :lname, :address, :salary
5) FROM EMPLOYEE WHERE Ssn = :ssn} ;
6) } catch (SQLException se) {
7) System.out.println("Social Security Number does not exist: " + ssn) ;
8) Return ;
9) }
10) System.out.println(fname + " " + minit + " " + lname + " " + address
+ " " + salary)
```

# Retrieving Multiple Tuples in SQLJ Using Iterators

## ■ **Iterator**

- Object associated with a collection (set or multiset) of records in a query result

## ■ **Named iterator**

- Associated with a query result by listing attribute names and types in query result

## ■ **Positional iterator**

- Lists only attribute types in query result

**Figure 10.8** Program segment J2A, a Java program segment that uses a **named iterator** to print employee information in a particular department

```
//Program Segment J2A:
0) dname = readEntry("Enter the Department Name: ") ;
1) try {
2) #sql { SELECT Dnumber INTO :dnumber
3) FROM DEPARTMENT WHERE Dname = :dname} ;
4) } catch (SQLException se) {
5) System.out.println("Department does not exist: " + dname) ;
6) Return ;
7) }
8) System.out.println("Employee information for Department: " + dname) ;
9) #sql iterator Emp(String ssn, String fname, String minit, String lname,
 double salary) ;
10) Emp e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12) FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) while (e.next()) {
14) System.out.println(e.ssn + " " + e.fname + " " + e.minit + " "
 +
 e.lname + " " + e.salary) ;
15) } ;
16) e.close() ;
```

**Figure 10.9** Program segment J2B, a Java program segment that uses a **positional iterator** to print employee information in a particular department.

```
//Program Segment J2B:
0) dname = readEntry("Enter the Department Name: ") ;
1) try {
2) #sql { SELECT Dnumber INTO :dnumber
3) FROM DEPARTMENT WHERE Dname = :dname} ;
4) } catch (SQLException se) {
5) System.out.println("Department does not exist: " + dname) ;
6) Return ;
7) }
8) System.out.printline("Employee information for Department: " + dname) ;
9) #sql iterator Empos(String, String, String, String, double) ;
10) Empos e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12) FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15) System.out.printline(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16) #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```

# Database Programming with Function Calls: SQL/CLI & JDBC

- Use of function calls
  - **Dynamic** approach for database programming
- Library of functions
  - Also known as **application programming interface (API)**
  - Used to access database
- **SQL Call Level Interface (SQL/CLI)**
  - Part of SQL standard

# SQL/CLI: Using C as the Host Language

- **Environment record**
  - Track one or more database connections
  - Set environment information
- **Connection record**
  - Keeps track of information needed for a particular database connection
- **Statement record**
  - Keeps track of the information needed for one SQL statement

# SQL/CLI: Using C as the Host Language (cont'd.)

- **Description record**
  - Keeps track of information about tuples or parameters
- **Handle to the record**
  - C pointer variable makes record accessible to program

**Figure**

```
//Program CLI1:
0) #include sqlcli.h ;
1) void printSal() {
2) SQLHSTMT stmt1 ;
3) SQLHDBC con1 ;
4) SQLHENV env1 ;
5) SQLRETURN ret1, ret2, ret3, ret4 ;
6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
 SQL_NTS) else exit ;
9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Ssn = ?",
 SQL_NTS) ;
11) prompt("Enter a Social Security Number: ", ssn) ;
12) SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15) SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16) SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17) ret2 = SQLFetch(stmt1) ;
18) if (!ret2) printf(ssn, lname, salary)
19) else printf("Social Security Number does not exist: ", ssn) ;
20) }
21) }
```

**Figure 10.11** Program segment CLI2, a C program segment that uses SQL/CLI for a query with a collector

```
//Program Segment CLI2:
0) #include sqlcli.h ;
1) void printDepartmentEmps() {
2) SQLHSTMT stmt1 ;
3) SQLHDBC con1 ;
4) SQLHENV env1 ;
5) SQLRETURN ret1, ret2, ret3, ret4 ;
6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
 SQL_NTS) else exit ;
9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Dno = ?",
 SQL_NTS) ;
11) prompt("Enter the Department Number: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15) SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16) SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17) ret2 = SQLFetch(stmt1) ;
18) while (!ret2) {
19) printf(lname, salary) ;
20) ret2 = SQLFetch(stmt1) ;
21) }
22) }
23) }
```

# JDBC: SQL Function Calls for Java Programming

- JDBC
  - Java function libraries
- Single Java program can connect to several different databases
  - Called data sources accessed by the Java program
- `Class.forName("oracle.jdbc.driver.OracleDriver")`
  - Load a **JDBC driver** explicitly

# JDBC: SQL Function Calls for Java Programming

- **Connection object**
- **Statement object** has two subclasses:
  - PreparedStatement and CallableStatement
- Question mark (?) symbol
  - Represents a statement parameter
  - Determined at runtime
- **ResultSet object**
  - Holds results of query

**Figure 10**

```
//Program JDBC1:
0) import java.io.* ;
1) import java.sql.*
...
2) class getEmpInfo {
3) public static void main (String args []) throws SQLException, IOException {
4) try { Class.forName("oracle.jdbc.driver.OracleDriver")
5) } catch (ClassNotFoundException x) {
6) System.out.println ("Driver could not be loaded") ;
7) }
8) String dbacct, passwd, ssn, lname ;
9) Double salary ;
10) dbacct = readentry("Enter database account:") ;
11) passwd = readentry("Enter password:") ;
12) Connection conn = DriverManager.getConnection
13) ("jdbc:oracle:oci8:" + dbacct + "/" + passwd) ;
14) String stmt1 = "select Lname, Salary from EMPLOYEE where Ssn = ?" ;
15) PreparedStatement p = conn.prepareStatement(stmt1) ;
16) ssn = readentry("Enter a Social Security Number: ") ;
17) p.clearParameters() ;
18) p.setString(1, ssn) ;
19) ResultSet r = p.executeQuery() ;
20) while (r.next()) {
21) lname = r.getString(1) ;
22) salary = r.getDouble(2) ;
23) System.out.println(lname + salary) ;
24) } }
25) }
```

**Figure 10.13** Program segment JDBC2, a Java program segment that uses JDBC for a query with

```
//Program Segment JDBC2:
0) import java.io.* ;
1) import java.sql.*
 ...
2) class printDepartmentEmps {
3) public static void main (String args [])
 throws SQLException, IOException {
4) try { Class.forName("oracle.jdbc.driver.OracleDriver")
5) } catch (ClassNotFoundException x) {
6) System.out.println ("Driver could not be loaded") ;
7) }
8) String dbacct, passwd, lname ;
9) Double salary ;
10) Integer dno ;
11) dbacct = readentry("Enter database account:") ;
12) passwd = readentry("Enter password:") ;
13) Connection conn = DriverManager.getConnection
14) ("jdbc:oracle:oci8:" + dbacct + "/" + passwd) ;
15) dno = readentry("Enter a Department Number: ") ;
16) String q = "select Lname, Salary from EMPLOYEE where Dno = " +
17) dno.toString() ;
18) Statement s = conn.createStatement() ;
19) ResultSet r = s.executeQuery(q) ;
20) while (r.next()) {
21) lname = r.getString(1) ;
22) salary = r.getDouble(2) ;
23) System.out.println(lname + salary) ;
24) }
```

# Database Stored Procedures and SQL/PSM

## ■ **Stored procedures**

- Program modules stored by the DBMS at the database server
- Can be functions or procedures

## ■ **SQL/PSM (**SQL/Persistent Stored Modules**)**

- Extensions to SQL
- Include general-purpose programming constructs in SQL

# Database Stored Procedures and Functions

- **Persistent stored modules**
  - Stored persistently by the DBMS
- Useful:
  - When database program is needed by several applications
  - To reduce data transfer and communication cost between client and server in certain situations
  - To enhance modeling power provided by views

# Database Stored Procedures and Functions (cont'd.)

## ■ Declaring stored procedures:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;
declaring a function, a return type is necessary,
so the declaration form is
```

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

# Database Stored Procedures and Functions (cont'd.)

- Each parameter has parameter type
  - **Parameter type:** one of the SQL data types
  - **Parameter mode:** IN, OUT, or INOUT
- Calling a stored procedure:  
`CALL <procedure or function name>  
(<argument list>) ;`

# SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

- Conditional branching statement:

```
IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
 ...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;
```

# SQL/PSM (cont'd.)

## ■ Constructs for looping

```
WHILE <condition> DO
 <statement list>
END WHILE ;
REPEAT
 <statement list>
UNTIL <condition>
END REPEAT ;
```

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
 <statement list>
END FOR ;
```

**Figure 10.14** Declaring a function in SQL/PSM.

```
//Function PSM1:
0) CREATE FUNCTION Dept_size(IN deptno INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE No_of_emps INTEGER ;
3) SELECT COUNT(*) INTO No_of_emps
4) FROM EMPLOYEE WHERE Dno = deptno ;
5) IF No_of_emps > 100 THEN RETURN "HUGE"
6) ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7) ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8) ELSE RETURN "SMALL"
9) END IF ;
```

# Comparing the Three Approaches

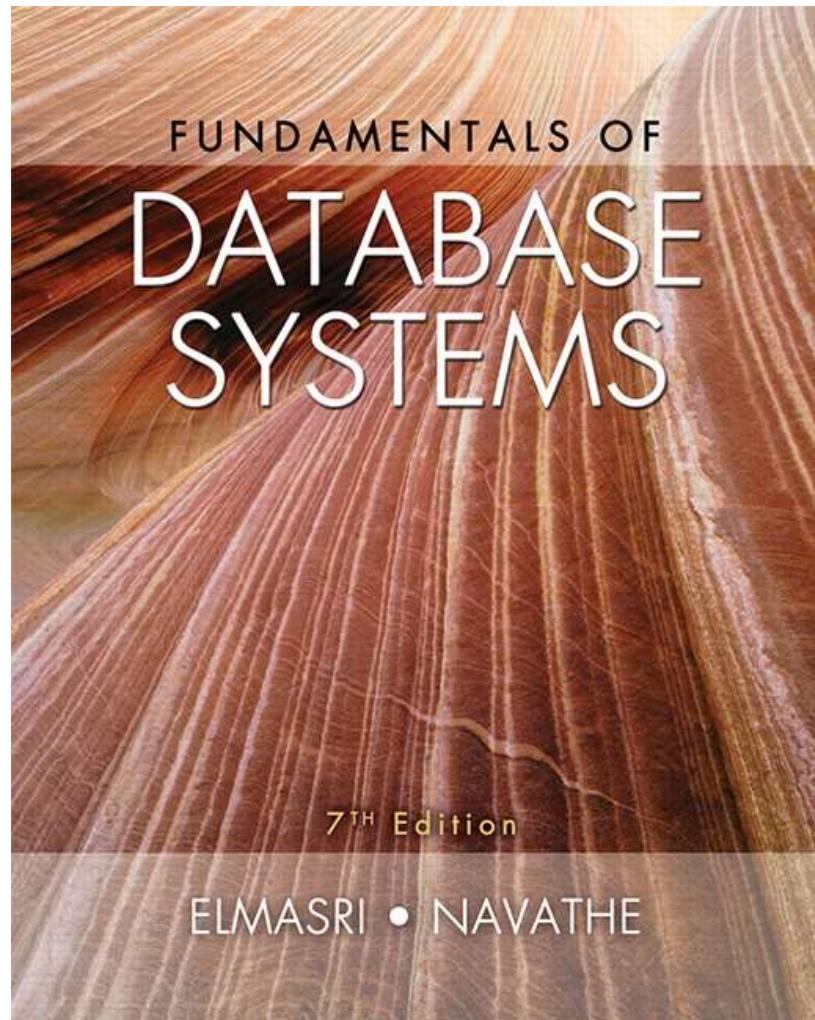
- Embedded SQL Approach
  - Query text checked for syntax errors and validated against database schema at compile time
  - For complex applications where queries have to be generated at runtime
    - Function call approach more suitable

# Comparing the Three Approaches (cont'd.)

- Library of Function Calls Approach
  - More flexibility
  - More complex programming
  - No checking of syntax done at compile time
- Database Programming Language Approach
  - Does not suffer from the impedance mismatch problem
  - Programmers must learn a new language

# Summary

- Techniques for database programming
  - Embedded SQL
  - SQLJ
  - Function call libraries
  - SQL/CLI standard
  - JDBC class library
  - Stored procedures
  - SQL/PSM



# Chapter 11 Outline

- A Simple PHP Example
- Overview of Basic Features of PHP
- Overview of PHP Database Programming

# Web Database Programming Using PHP

- Techniques for programming dynamic features into Web
- PHP
  - Open source scripting language
  - Interpreters provided free of charge
  - Available on most computer platforms

# A Simple PHP Example

- PHP

- Open source general-purpose scripting language
- Comes installed with the UNIX operating system

# A Simple PHP Example (cont'd.)

- DBMS
  - **Bottom-tier database server**
- PHP
  - **Middle-tier Web server**
- HTML
  - **Client tier**

**Figure 11.1a** PHP program segment for entering a greeting.

(a)

```
//Program Segment P1:
0) <?php
1) // Printing a welcome message if the user submitted their name
 // through the HTML form
2) if ($_POST['user_name']) {
3) print("Welcome, ");
4) print($_POST['user_name']);
5) }
6) else {
7) // Printing the form to enter the user name since no name has
 // been entered yet
8) print <<<_HTML_
9) <FORM method="post" action="$_SERVER['PHP_SELF']">
10) Enter your name: <input type="text" name="user_name">
11)

12) <INPUT type="submit" value="SUBMIT NAME">
13) </FORM>
14) _HTML_;
15) }
16) ?>
```

*continued on next slide*

**Figure 11.1b-d** (b) Initial form displayed by PHP program segment. (c) User enters name *John Smith*. (d) Form prints welcome message for *John Smith*.

(b)

Enter your name:

**SUBMIT NAME**

(c)

Enter your name:

**SUBMIT NAME**

(d)

Welcome, John Smith

# A Simple PHP Example (cont'd.)

- Example Figure 11.1(a)
- PHP script stored in:
  - <http://www.myserver.com/example/greeting.php>
- <?php
  - PHP start tag
- ?>
  - PHP end tag
- Comments: // or /\* \*/

# A Simple PHP Example (cont'd.)

- `$_POST`
  - **Auto-global** predefined PHP variable
  - Array that holds all the values entered through form parameters
- Arrays are dynamic
- **Long text strings**
  - Between opening `<<<__HTML__` and closing `_HTML_;`

# A Simple PHP Example (cont'd.)

- **PHP variable names**
  - Start with \$ sign

# Overview of Basic Features of PHP

- Illustrate features of PHP suited for creating dynamic Web pages that contain database access commands

# PHP Variables, Data Types, and Programming Constructs

- PHP variable names
  - Start with \$ symbol
  - Can include characters, letters, and underscore character (\_)
- Main ways to express strings and text
  - Single-quoted strings
  - Double-quoted strings
  - Here documents
  - Single and double quotes

# PHP Variables, Data Types, and Programming Constructs (cont'd.)

## ■ Period (.) symbol

- String concatenate operator
- Single-quoted strings
  - Literal strings that contain no PHP program variables
- Double-quoted strings and here documents
  - Values from variables need to be interpolated into string

# PHP Variables, Data Types, and Programming Constructs (cont'd.)

- Numeric data types
  - Integers and floating points
- Programming language constructs
  - For-loops, while-loops, and conditional if-statements
- Boolean expressions

**Figure 11.2** Illustrating basic PHP string and text values.

```
0) print 'Welcome to my Web site.';
1) print 'I said to him, "Welcome Home"';
2) print 'We\'ll now visit the next Web site';
3) printf('The cost is $%.2f and the tax is $%.2f',
 $cost, $tax);
4) print strtolower('AbCdE');
5) print ucwords(strtolower('JOHN smith'));
6) print 'abc' . 'efg'
7) print "send your email reply to: $email_address"
8) print <<<FORM_HTML
9) <FORM method="post" action="$_SERVER['PHP_SELF']">
10) Enter your name: <input type="text" name="user_name">
11) FORM_HTML
```

# PHP Variables, Data Types, and Programming Constructs (cont'd.)

- Comparison operators
  - == (equal), != (not equal), > (greater than), >= (greater than or equal), < (less than), and <= (less than or equal)

# PHP Arrays

- Can hold database query results
  - Two-dimensional arrays
  - First dimension representing rows of a table
  - Second dimension representing columns (attributes) within a row
- Main types of arrays:
  - **Numeric** and **associative**

# PHP Arrays (cont'd.)

- Numeric array
  - Associates a numeric index with each element in the array
  - Indexes are integer numbers
    - Start at zero
    - Grow incrementally
- Associative array
  - Provides pairs of (key => value) elements

**Figure 11.3** Illustrating basic PHP array processing.

```
0) $teaching = array('Database' => 'Smith', 'OS' => 'Carrick',
 'Graphics' => 'Kam');
1) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Li';
2) sort($teaching);
3) foreach ($teaching as $key => $value) {
4) print " $key : $value\n";
5) $courses = array('Database', 'OS', 'Graphics', 'Data Mining');
6) $alt_row_color = array('blue', 'yellow');
7) for ($i = 0, $num = count($courses); i < $num; $i++) {
8) print '<TR bgcolor="' . $alt_row_color[$i % 2] . '">';
9) print "<TD>Course $i is</TD><TD>$course[$i]</TD></TR>\n";
10) }
```

# PHP Arrays (cont'd.)

- Techniques for looping through arrays in PHP
- Count function
  - Returns current number of elements in array
- Sort function
  - Sorts array based on element values in it

# PHP Functions

- Functions
  - Define to structure a complex program and to share common sections of code
  - Arguments passed by value
- Examples to illustrate basic PHP functions
  - Figure 11.4
  - Figure 11.5

**Figure 11.4**

```
//Program Segment P1':
0) function display_welcome() {
1) print("Welcome, ");
2) print($_POST['user_name']);
3)
4)
5) function display_empty_form(); {
6) print <<<_HTML_
7) <FORM method="post" action="$_SERVER['PHP_SELF']">
8) Enter your name: <INPUT type="text" name="user_name">
9)

10) <INPUT type="submit" value="Submit name">
11) </FORM>
12) _HTML_;
13)
14) if ($_POST['user_name']) {
15) display_welcome();
16)
17) else {
18) display_empty_form();
19) }
```

**Figure 11.5** Illustrating a function with arguments and return value.

```
0) function course_instructor ($course, $teaching_assignments) {
1) if (array_key_exists($course, $teaching_assignments)) {
2) $instructor = $teaching_assignments[$course];
3) RETURN "$instructor is teaching $course";
4) }
5) else {
6) RETURN "there is no $course course";
7) }
8) }
9) $teaching = array('Database' => 'Smith', 'OS' => 'Carrick',
 'Graphics' => 'Kam');
10) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Li';
11) $x = course_instructor('Database', $teaching);
12) print($x);
13) $x = course_instructor('Computer Architecture', $teaching);
14) print($x);
```

# PHP Server Variables and Forms

- Built-in entries

- `$_SERVER` auto-global built-in array variable
- Provides useful information about server where the PHP interpreter is running

# PHP Server Variables and Forms (cont'd.)

- Examples:
  - `$_SERVER['SERVER_NAME']`
  - `$_SERVER['REMOTE_ADDRESS']`
  - `$_SERVER['REMOTE_HOST']`
  - `$_SERVER['PATH_INFO']`
  - `$_SERVER['QUERY_STRING']`
  - `$_SERVER['DOCUMENT_ROOT']`
- `$_POST`
  - Provides input values submitted by the user through HTML forms specified in `<INPUT>` tag

# Overview of PHP Database Programming

- PEAR DB library
  - Part of PHP Extension and Application Repository (PEAR)
  - Provides functions for database access

# Connecting to a Database

- Library module DB.php must be loaded
- DB library functions accessed using

DB::<function\_name>

- DB::connect ('string')
  - Function for connecting to a database
  - Format for 'string' is: <DBMS software>://<user account>:<password>@<database server>

# Figure 11-8 Connecting to a database, creating a table, and inserting a record.

```
0) require 'DB.php';
1) $d = DB::connect('oci8://acct1:pass12@www.host.com/db1');
2) if (DB::isError($d)) { die("cannot connect - " . $d->getMessage()); }
 ...
3) $q = $d->query("CREATE TABLE EMPLOYEE
4) (Emp_id INT,
5) Name VARCHAR(15),
6) Job VARCHAR(10),
7) Dno INT);");
8) if (DB::isError($q)) { die("table creation not successful - " .
 $q->getMessage()); }

 ...
9) $d->setErrorHandler(PEAR_ERROR_DIE);
 ...
10) $eid = $d->nextID('EMPLOYEE');
11) $q = $d->query("INSERT INTO EMPLOYEE VALUES
12) ($eid, $_POST['emp_name'], $_POST['emp_job'], $_POST['emp_dno'])");
 ...
13) $eid = $d->nextID('EMPLOYEE');
14) $q = $d->query('INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?, ?)',
15) array($eid, $_POST['emp_name'], $_POST['emp_job'], $_POST['emp_dno'])');
```

# Some Database Functions

- Query function
  - `$d->query` takes an SQL command as its string argument
  - Sends query to database server for execution
- `$d->setErrorHandler(PEAR_ERROR_DIE)`
  - Terminate program and print default error messages if any subsequent errors occur

# Collecting Data from Forms and Inserting Records

- Collect information through HTML or other types of Web forms
- Create unique record identifier for each new record inserted into the database
- PHP has a function `$d->nextID` to create a sequence of unique values for a particular table
- **Placeholders**
  - Specified by ? symbol

# Retrieval Queries from Database Tables

- \$q
  - Variable that holds query result
  - \$q->fetchRow () retrieve next record in query result and control loop
- \$allresult = \$d->getAll (query)
  - Holds all the records in a query result in a single variable called \$allresult

## Figure

```
0) require 'DB.php';
1) $d = DB::connect('oci8://acct1:pass12@www.host.com/dbname');
2) if (DB::isError($d)) { die("cannot connect - " . $d->getMessage()); }
3) $d->setErrorHandler(PEAR_ERROR_DIE);
...
4) $q = $d->query('SELECT Name, Dno FROM EMPLOYEE');
5) while ($r = $q->fetchRow()) {
6) print "employee $r[0] works for department $r[1] \n" ;
7) }
...
8) $q = $d->query('SELECT Name FROM EMPLOYEE WHERE Job = ? AND Dno = ?',
9) array($_POST['emp_job'], $_POST['emp_dno']));
10) print "employees in dept $_POST['emp_dno'] whose job is
 $_POST['emp_job']: \n"
11) while ($r = $q->fetchRow()) {
12) print "employee $r[0] \n" ;
13) }
...
14) $allresult = $d->getAll('SELECT Name, Job, Dno FROM EMPLOYEE');
15) foreach ($allresult as $r) {
16) print "employee $r[0] has job $r[1] and works for department $r[2] \n" ;
17) }
...
...
```

# Other techniques

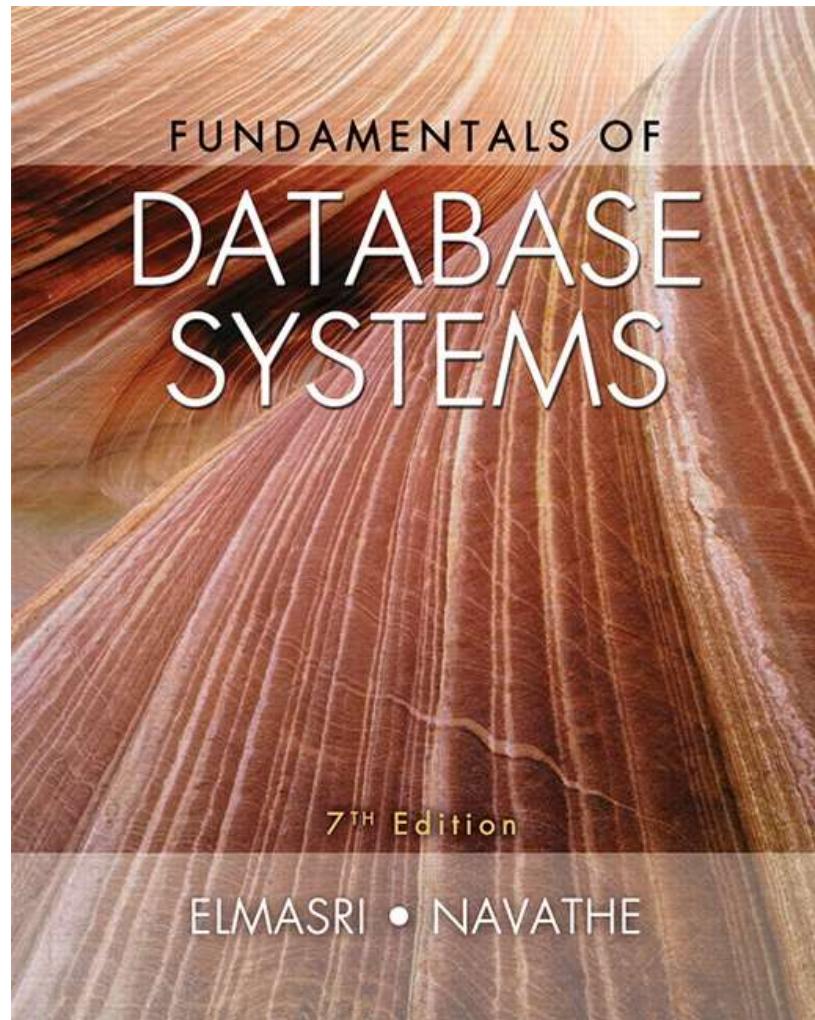
- PHP runs on server
  - Sends HTML to client
- Many other languages/technologies for Web Db programming
- Examples:
- Java servlets:
  - Java objects on server, interact with client
  - Store information about interaction session

# Other techniques (cont.)

- Java Server Pages (JSP)
  - Creates dynamic Web pages through scripting at server to send to client (somewhat like PHP)
- JavaScript
  - Scripting language, can run at client or server
- Java Script Object Notation (JSON):
  - Text-based representation of objects
  - Similar function to XML
  - Used in many NOSQL systems

# Summary

- PHP scripting language
  - Very popular for Web database programming
- PHP basics for Web programming
- Data types
- Database commands include:
  - Creating tables, inserting new records, and retrieving database records
  - Looping over a query result



# Chapter 12 Outline

- Overview of Object Database Concepts
- Object-Relational Features
- Object Database Extensions to SQL
- ODMG Object Model and the Object Definition Language ODL
- Object Database Conceptual Design
- The Object Query Language OQL
- Overview of the C++ Language Binding

# Object and Object-Relational Databases

- **Object databases (ODB)**
  - **Object data management systems (ODMS)**
  - Meet some of the needs of more complex applications
  - Specify:
    - Structure of complex objects
    - Operations that can be applied to these objects

# Overview of Object Database Concepts

- Introduction to object-oriented concepts and features
  - Origins in OO programming languages
  - Object has two components:
    - State (value) and behavior (operations)
  - Instance variables (attributes)
    - Hold values that define internal state of object
  - Operation is defined in two parts:
    - Signature (interface) and implementation (method)

# Overview of Object Database Concepts (cont'd.)

- Inheritance
  - Permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes
- Operator overloading
  - Operation's ability to be applied to different types of objects
  - Operation name may refer to several distinct implementations

# Object Identity, and Objects versus Literals

- Object has Unique identity
  - Implemented via a unique, system-generated object identifier (OID)
  - **Immutable**
- Most OO database systems allow for the representation of both objects and literals (simple or complex values)

# Complex Type Structures for Objects and Literals

- Structure of arbitrary complexity
  - Contain all necessary information that describes object or literal
- Nesting **type constructors**
  - Generate complex type from other types
- Type constructors (type generators):
  - **Atom (basic data type – int, string, etc.)**
  - **Struct (or tuple)**
  - **Collection**

# Complex Type Structures for Objects and Literals (cont'd.)

- Collection types:
  - **Set**
  - **Bag**
  - **List**
  - **Array**
  - **Dictionary**
- **Object definition language (ODL)**
  - Used to define object types for a particular database application

**Figure 12.1** Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors

```
define type EMPLOYEE
 tuple (Fname: string;
 Minit : char;
 Lname: string;
 Ssn: string;
 Birth_date: DATE;
 Address: string;
 Sex: char;
 Salary: float;
 Supervisor: EMPLOYEE;
 Dept: DEPARTMENT);

define type DATE
 tuple (Year: integer;
 Month: integer;
 Day: integer;);

define type DEPARTMENT
 tuple (Dname: string;
 Dnumber: integer;
 Mgr: tuple (Manager: EMPLOYEE;
 Start_date: DATE;);
 Locations: set(string);
 Employees: set(EMPLOYEE);
 Projects: set(PROJECT););
```

**Figure 12.2** Adding op

```
define class EMPLOYEE
 type tuple (Fname: string;
 Minit: char;
 Lname: string;
 Ssn: string;
 Birth_date: DATE;
 Address: string;
 Sex: char;
 Salary: float;
 Supervisor: EMPLOYEE;
 Dept: DEPARTMENT;);
 operations
 age: integer;
 create_emp: EMPLOYEE;
 destroy_emp: boolean;
 end EMPLOYEE;
define class DEPARTMENT
 type tuple (Dname: string;
 Dnumber: integer;
 Mgr: tuple (Manager: EMPLOYEE;
 Start_date: DATE;);
 Locations: set (string);
 Employees: set (EMPLOYEE);
 Projects: set (PROJECT););
 operations
 no_of_emps: integer;
 create_dept: DEPARTMENT;
 destroy_dept: boolean;
 assign_emp(e: EMPLOYEE): boolean;
 (* adds an employee to the department *)
 remove_emp(e: EMPLOYEE): boolean;
 (* removes an employee from the department *)
 end DEPARTMENT;
```

DEPARTMENT.

# Encapsulation of Operations

- Encapsulation
  - Related to abstract data types
  - Define **behavior** of a class of object based on operations that can be externally applied
  - External users only aware of interface of the operations
  - Can divide structure of object into visible and hidden attributes

# Encapsulation of Operations

- **Constructor** operation
  - Used to create a new object
- **Destructor** operation
  - Used to destroy (delete) an object
- **Modifier** operations
  - Modify the state of an object
- **Retrieve** operation
- *Dot notation* to apply operations to object

# Persistence of Objects

## ■ Transient objects

- Exist in executing program
- Disappear once program terminates

## ■ Persistent objects

- Stored in database, persist after program termination
- **Naming mechanism:** object assigned a unique name in object base, user finds object by its name
- **Reachability:** object referenced from other persistent objects, object located through references

## Figures

```
define class DEPARTMENT_SET
 type set (DEPARTMENT);
 operations add_dept(d: DEPARTMENT): boolean;
 (* adds a department to the DEPARTMENT_SET object *)
 remove_dept(d: DEPARTMENT): boolean;
 (* removes a department from the DEPARTMENT_SET object *)
 create_dept_set: DEPARTMENT_SET;
 destroy_dept_set: boolean;
end Department_Set;

...
persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
(* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
...
d:= create_dept;
(* create a new DEPARTMENT object in the variable d *)
...
b:= ALL_DEPARTMENTS.add_dept(d);
(* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

# Type (Class) Hierarchies and Inheritance

- Inheritance
  - Definition of new types based on other predefined types
  - Leads to **type (or class) hierarchy**
- Type: **type name** and list of visible (public) **functions** (attributes or operations)
  - Format:
    - `TYPE_NAME: function, function, . . . , function`

# Type (Class) Hierarchies and Inheritance (cont'd.)

## ■ Subtype

- Useful when creating a new type that is similar but not identical to an already defined type
- Subtype inherits functions
- Additional (local or specific) functions in subtype
- Example:
  - EMPLOYEE subtype-of PERSON: Salary, Hire\_date, Seniority
  - STUDENT subtype-of PERSON: Major, Gpa

# Type (Class) Hierarchies and Inheritance (cont'd.)

## ■ Extent

- A *named persistent object* to hold collection of all persistent objects for a class

## ■ Persistent collection

- Stored permanently in the database

## ■ Transient collection

- Exists temporarily during the execution of a program (e.g. query result)

# Other Object-Oriented Concepts

- **Polymorphism** of operations
  - Also known as **operator overloading**
  - Allows same operator name or symbol to be bound to two or more different implementations
  - Type of objects determines which operator is applied
- **Multiple inheritance**
  - Subtype inherits functions (attributes and operations) of more than one supertype

# Summary of Object Database Concepts

- Object identity
- Type constructors (type generators)
- Encapsulation of operations
- Programming language compatibility
- Type (class) hierarchies and inheritance
- Extents
- Polymorphism and operator overloading

# Object-Relational Features: Object DB Extensions to SQL

- **Type constructors (generators)**
  - Specify complex types using UDT
- Mechanism for specifying **object identity**
- **Encapsulation of operations**
  - Provided through user-defined types (UDTs)
- **Inheritance** mechanisms
  - Provided using keyword UNDER

# User-Defined Types (UDTs) and Complex Structures for Objects

## ■ UDT syntax:

- CREATE TYPE <type name> AS (<component declarations>);
- Can be used to create a complex type for an attribute (similar to *struct* – no operations)
- Or: can be used to create a type as a basis for a table of objects (similar to *class* – can have operations)

# User-Defined Types and Complex Structures for Objects (cont'd.)

- Array type – to specify collections
  - Reference array elements using []
- **CARDINALITY** function
  - Return the current number of elements in an array
- Early SQL had only array for collections
  - Later versions of SQL added other collection types (set, list, bag, array, etc.)

# Object Identifiers Using Reference Types

## ■ Reference type

- Create unique object identifiers (OIDs)
- Can specify system-generated object identifiers
- Alternatively can use primary key as OID as in traditional relational model
- Examples:
  - REF IS SYSTEM GENERATED
  - REF IS <OID\_ATTRIBUTE>  
<VALUE\_GENERATION\_METHOD> ;

# Creating Tables Based on the UDTs

## ■ INSTANTIABLE

- Specify that UDT is instantiable
- The user can then create one or more tables based on the UDT
- If keyword INSTANTIABLE is left out, can use UDT only as attribute data type – not as a basis for a table of objects

# Encapsulation of Operations

- User-defined type
  - Specify methods (or operations) in addition to the attributes
  - Format:

```
CREATE TYPE <TYPE-NAME> (
 <LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>
 <DECLARATION OF FUNCTIONS (METHODS)>
) ;
```

**Figure 12.4a** Illustrating some of the object features of SQL. Using UDTs as types for attributes such as Address and Phone.

```
(a) CREATE TYPE STREET_ADDR_TYPE AS (
 NUMBER VARCHAR (5),
 STREET NAME VARCHAR (25),
 APT_NO VARCHAR (5),
 SUITE_NO VARCHAR (5)
);
CREATE TYPE USA_ADDR_TYPE AS (
 STREET_ADDR STREET_ADDR_TYPE,
 CITY VARCHAR (25),
 ZIP VARCHAR (10)
);
CREATE TYPE USA_PHONE_TYPE AS (
 PHONE_TYPE VARCHAR (5),
 AREA_CODE CHAR (3),
 PHONE_NUM CHAR (7)
);
```

*continued on next slide*

**Figure 12.4b** Illustrating some of the object features of SQL. Specifying UDT for PERSON\_TYPE.

```
(b) CREATE TYPE PERSON_TYPE AS (
 NAME VARCHAR (35),
 SEX CHAR,
 BIRTH_DATE DATE,
 PHONES USA_PHONE_TYPE ARRAY [4],
 ADDR USA_ADDR_TYPE
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD AGE() RETURNS INTEGER;
CREATE INSTANCE METHOD AGE() RETURNS INTEGER
 FOR PERSON_TYPE
BEGIN
 RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
 TODAY'S DATE AND SELF.BIRTH_DATE */
END;
);
```

*continued on next slide*

# Specifying Type Inheritance

- NOT FINAL:
  - The keyword NOT FINAL indicates that subtypes can be created for that type
- UNDER
  - The keyword UNDER is used to create a subtype

**Figure 12.4c** Illustrating some of the object features of SQL. Specifying UDTs for STUDENT\_TYPE and EMPLOYEE\_TYPE as two subtypes of PERSON\_TYPE.

```
(c) CREATE TYPE GRADE_TYPE AS (
 COURSENO CHAR (8),
 SEMESTER VARCHAR (8),
 YEAR CHAR (4),
 GRADE CHAR
);
CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
 MAJOR_CODE CHAR (4),
 STUDENT_ID CHAR (12),
 DEGREE VARCHAR (5),
 TRANSCRIPT GRADE_TYPE ARRAY [100]
```

*continued on next slide*

**Figure 12.4c (continued)** Illustrating some of the object features of SQL.  
Specifying UDTs for STUDENT\_TYPE and EMPLOYEE\_TYPE as two subtypes of PERSON\_TYPE.

```
INSTANTIABLE
NOT FINAL
INSTANCE METHOD GPA() RETURNS FLOAT;
CREATE INSTANCE METHOD GPA() RETURNS FLOAT
FOR STUDENT_TYPE
BEGIN
 RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
 SELF.TRANSCRIPT */
END;
);
CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
 JOB_CODE CHAR (4),
 SALARY FLOAT,
 SSN CHAR (11)
INSTANTIABLE
NOT FINAL
);
CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
 DEPT_MANAGED CHAR (20)
INSTANTIABLE
);
```

*continued on next slide*

# Specifying Type Inheritance

- Type inheritance rules:
  - All attributes/operations are inherited
  - Order of supertypes in UNDER clause determines inheritance hierarchy
  - Instance (object) of a subtype can be used in every context in which a supertype instance used
  - Subtype can redefine any function defined in supertype

# Creating Tables based on UDT

- UDT must be INSTANTIABLE
- One or more tables can be created
- Table inheritance:
  - UNDER keyword can also be used to specify supertable/subtable inheritance
  - Objects in subtable must be a **subset of** the objects in the supertable

**Figure 12.4d** Illustrating some of the object features of SQL. Creating tables based on some of the UDTs, and illustrating table inheritance.

(d) **CREATE TABLE PERSON OF PERSON\_TYPE  
REF IS PERSON\_ID SYSTEM GENERATED;**  
**CREATE TABLE EMPLOYEE OF EMPLOYEE\_TYPE  
UNDER PERSON;**  
**CREATE TABLE MANAGER OF MANAGER\_TYPE  
UNDER EMPLOYEE;**  
**CREATE TABLE STUDENT OF STUDENT\_TYPE  
UNDER PERSON;**

*continued on next slide*

# Specifying Relationships via Reference

- Component attribute of one tuple may be a **reference** to a tuple of another table
  - Specified using keyword **REF**
- Keyword **SCOPE**
  - Specify name of table whose tuples referenced
- **Dot notation**
  - Build path expressions
- **->**
  - Used for dereferencing

**Figure 12.4e** Illustrating some of the object features of SQL. Specifying relationships using REF and SCOPE.

```
(e) CREATE TYPE COMPANY_TYPE AS (
 COMP_NAME VARCHAR (20),
 LOCATION VARCHAR (20));
CREATE TYPE EMPLOYMENT_TYPE AS (
 Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
 Company REF (COMPANY_TYPE) SCOPE (COMPANY));
CREATE TABLE COMPANY OF COMPANY_TYPE (
 REF IS COMP_ID SYSTEM GENERATED,
 PRIMARY KEY (COMP_NAME));
CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
```

# Summary of SQL Object Extensions

- UDT to specify complex types
  - INSTANTIABLE specifies if UDT can be used to create tables; NOT FINAL specifies if UDT can be inherited by a subtype
- REF for specifying **object identity** and inter-object references
- Encapsulation of operations in UDT
- Keyword UNDER to specify type inheritance and table inheritance

# ODMG Object Model and Object Definition Language ODL

- ODMG object model
  - Data model for **object definition language (ODL)** and **object query language (OQL)**
- Objects and Literals
  - Basic building blocks of the object model
- Object has five aspects:
  - **Identifier, name, lifetime, structure, and creation**
- Literal
  - Value that does not have an object identifier

# The ODMG Object Model and the ODL (cont'd.)

- **Behavior** refers to operations
- **State** refers to properties (attributes)
- **Interface**
  - Specifies only behavior of an object type
  - Typically **noninstantiable**
- **Class**
  - Specifies both state (attributes) and behavior (operations) of an object type
  - **Instantiable**

# Inheritance in the Object Model of ODMG

## ■ Behavior inheritance

- Also known as IS-A or interface inheritance
- Specified by the colon (:) notation

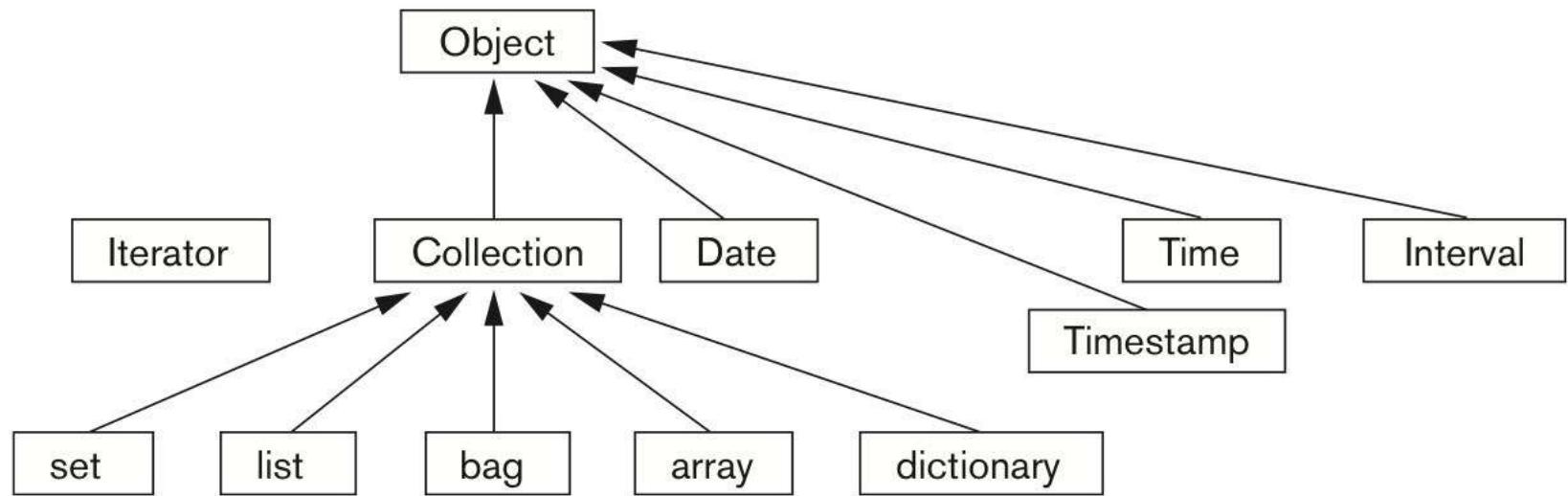
## ■ EXTENDS inheritance

- Specified by keyword **extends**
- Inherit both state and behavior strictly among classes
- Multiple inheritance via extends not permitted

# Built-in Interfaces and Classes in the Object Model

- **Collection objects**
  - Inherit the basic Collection interface
- `i = o.create_iterator()`
  - Creates an iterator object for the collection
  - To loop over each object in a collection
- **Collection objects further specialized into:**
  - set, list, bag, array, and dictionary

**Figure 12.6** Inheritance hierarchy for the built-in interfaces of the object model.



# Atomic (User-Defined) Objects

- Specified using keyword **class** in ODL
- **Attribute**
  - Property; describes data in an object
- **Relationship**
  - Specifies inter-object references
  - Keyword **inverse**
    - Single conceptual relationship in inverse directions
- **Operation signature:**
  - Operation name, argument types, return value

**Figure 12.7** The

```
class EMPLOYEE
(extent ALL_EMPLOYEES
 key Ssn)
{
 attribute string Name;
 attribute string Ssn;
 attribute date Birth_date;
 attribute enum Gender{M, F} Sex;
 attribute short Age;
 relationship DEPARTMENT Works_for
 inverse DEPARTMENT::Has_emps;
 void void reassign_emp(in string New_dname)
 raises(dname_not_valid);
};

class DEPARTMENT
(extent ALL_DEPARTMENTS
 key Dname, Dnumber)
{
 attribute string Dname;
 attribute short Dnumber;
 attribute struct Dept_mgr {EMPLOYEE Manager, date Start_date}
 Mgr;
 attribute set<string> Locations;
 attribute struct Projs {string Proj_name, time Weekly_hours}
 Projs;
 relationship set<EMPLOYEE> Has_emps inverse EMPLOYEE::Works_for;
 void void add_emp(in string New_ename) raises(ename_not_valid);
 void void change_manager(in string New_mgr_name; in date
 Start_date);
};
```

on.

# Extents, Keys, and Factory Objects

## ■ Extent

- A persistent named collection object that contains all persistent objects of class

## ■ Key

- One or more properties whose values are unique for each object in extent of a class

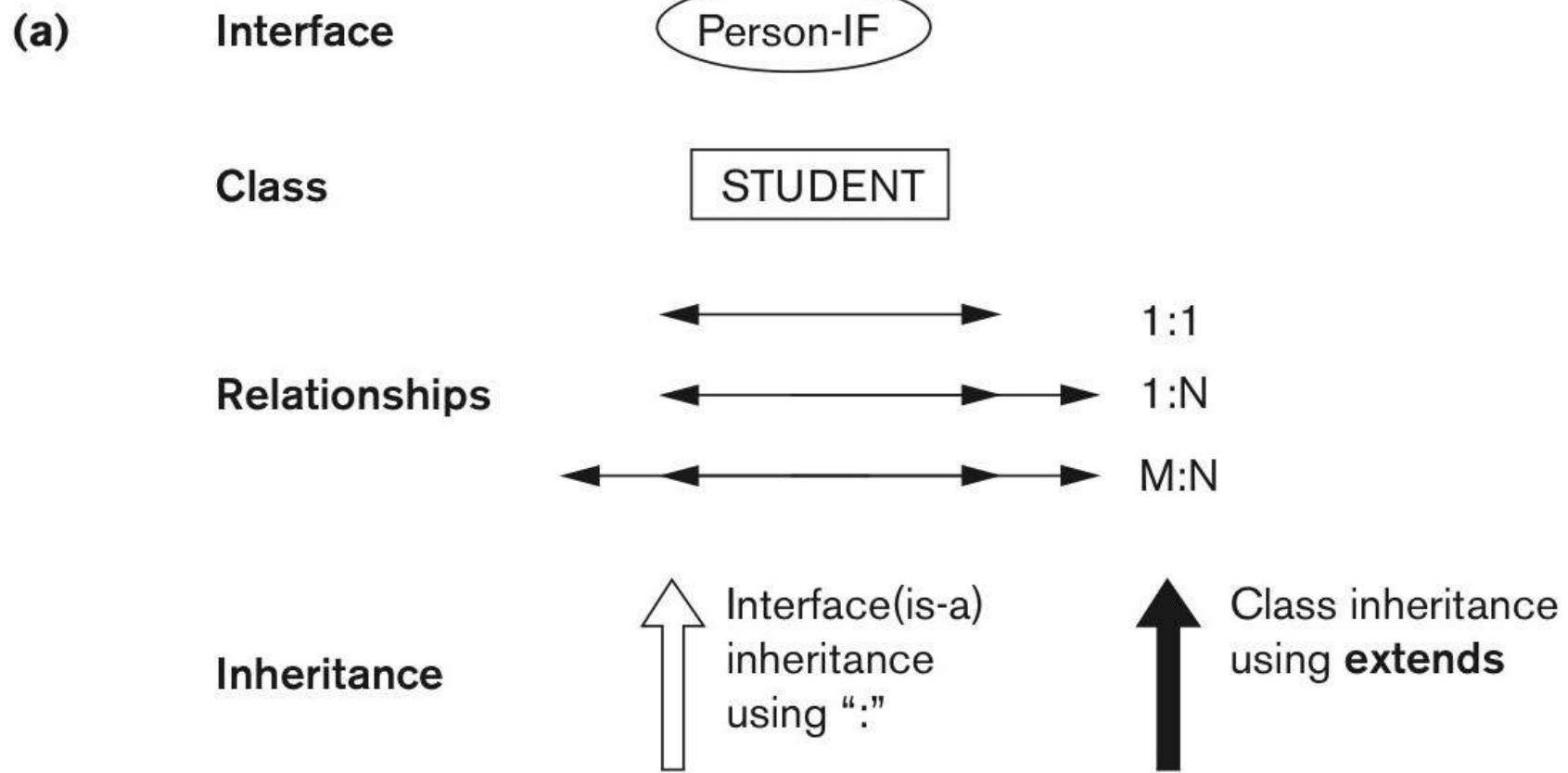
## ■ Factory object

- Used to generate or create individual objects via its operations

# Object Definition Language ODL

- Support semantic constructs of ODMG object model
- Independent of any particular programming language
- Example on next slides of a UNIVERSITY database
- Graphical diagrammatic notation is a variation of EER diagrams

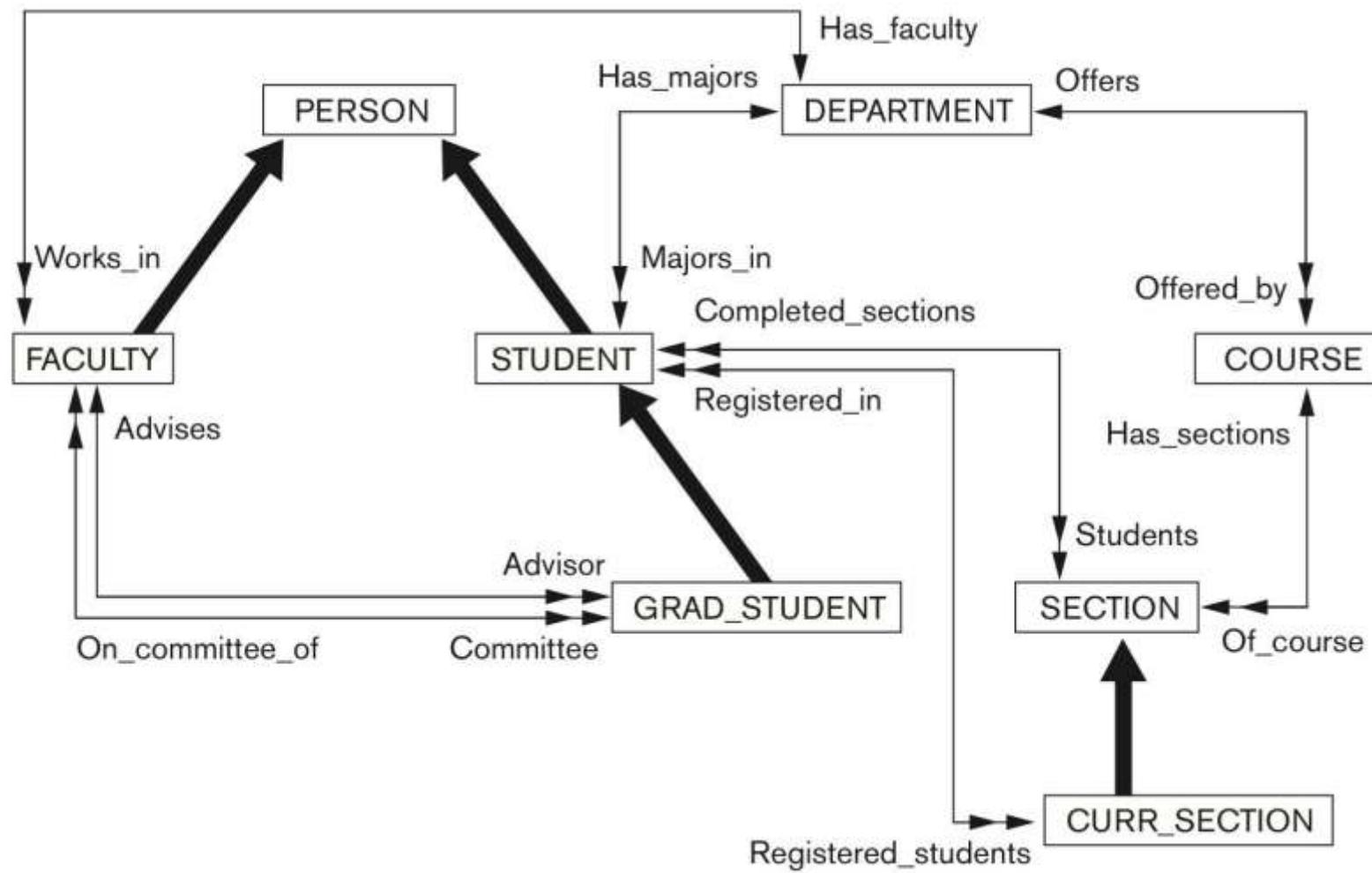
**Figure 12.9a** An example of a database schema. Graphical notation for representing ODL schemas.



*continued on next slide*

**Figure 12.9b** An example of a database schema. A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).

(b)



**Figure 12.10** Possible ODL schema for the UNIVERSITY database in Figure 12.9(b).

```

class PERSON
{
 extent PERSONS
 key Ssn
 attribute:
 struct Pname { string Fname,
 string Mname,
 string Lname } Name;
 string Ssn;
 date Birth_date;
 enum Gender{M, F} Sex;
 struct Address { short No,
 string Street,
 short Apt_no,
 string City,
 string State,
 short Zip } Address;
 short Age(); };
class FACULTY extends PERSON
{
 extent FACULTY
 attribute: string Rank;
 attribute: float Salary;
 attribute: string Office;
 attribute: string Phone;
 relationship DEPARTMENT Works_in inverse DEPARTMENT::Has_faculty;
 relationship set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
 relationship set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
 void give_raise(in float raise);
 void promote(in string new rank); };
class GRADE
{
 extent GRADES
 {
 attribute: enum GradeValues{A,B,C,D,F,I,P} Grade;
 relationship SECTION Section inverse SECTION::Students;
 relationship STUDENT Student inverse STUDENT::Completed_sections; };
class STUDENT extends PERSON
{
 extent STUDENTS
 attribute: string Class;
 attribute: Department Minors_in;
 relationship Department Majors_in inverse DEPARTMENT::Has_majors;
 relationship set<GRADE> Completed_sections inverse GRADE::Student;
 relationship set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
 void change_major(in string dname) raises(dname_not_valid);
 float gpa();
 void register(in short secno) raises(section_not_valid);
 void assign_grade(in short secno; IN GradeValue grade)
 raises(section_not_valid,grade_not_valid); };

```

*continued on next slide*

**Figure 12.10 (continued)** Possible ODL schema for the UNIVERSITY database in Figure 12.9(b).

```

class DEGREE
{
 attribute string College;
 attribute string Degree;
 attribute string Year;
};

class GRAD_STUDENT extends STUDENT
{
 extent GRAD_STUDENTS;
 attribute set<Degree> Degrees;
 relationship Faculty advisor inverse FACULTY::Advises;
 relationship set<FACULTY> Committee inverse FACULTY::On_committee_of;
 void assign_advisor(in string Lname; in string Fname)
 raises(faculty_not_valid);
 void assign_committee_member(in string Lname; in string Fname)
 raises(faculty_not_valid);
};

class DEPARTMENT
{
 extent DEPARTMENTS;
 key Dname;
 attribute string Dname;
 attribute string Dphone;
 attribute string Doffice;
 attribute string College;
 attribute FACULTY Chair;
 relationship set<FACULTY> Has_faculty inverse FACULTY::Works_in;
 relationship set<STUDENT> Has_majors inverse STUDENT::Majors_in;
 relationship set<COURSE> Offers inverse COURSE::Offered_by;
};

class COURSE
{
 extent COURSES;
 key Cno;
 attribute string Cname;
 attribute string Cno;
 attribute string Description;
 relationship set<SECTION> Has_sections inverse SECTION::Of_course;
 relationship <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers;
};

class SECTION
{
 extent SECTION;
 attribute short Sec_no;
 attribute string Year;
 attribute enum Quarter{Fall, Winter, Spring, Summer}
 Qtr;
 relationship set<Grade> Students inverse Grade::Section;
 relationship COURSE Of_course inverse COURSE::Has_sections;
};

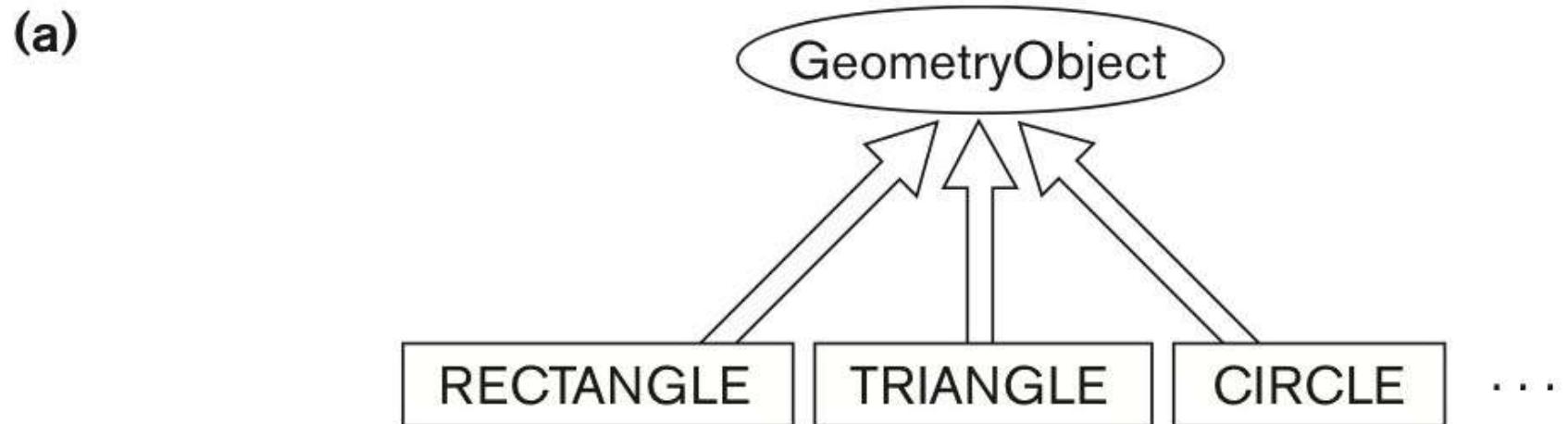
class CURR_SECTION extends SECTION
{
 extent CURRENT_SECTIONS;
 relationship set<STUDENT> Registered_students
 inverse STUDENT::Registered_in;
 void register_student(in string Ssn)
 raises(student_not_valid, section_full);
};

```

# Interface Inheritance in ODL

- Next example illustrates interface inheritance in ODL

**Figure 12.11a** An illustration of interface inheritance via “:”. Graphical schema representation.



*continued on next slide*

**Figure 12.11b** An illustration of interface inheritance via “:”. Corresponding interface and class definitions.

```
(b) interface GeometryObject
{ attribute enum Shape{RECTANGLE, TRIANGLE, CIRCLE, ... }
 attribute struct Point {short x, short y} Reference_point;
 float perimeter();
 float area();
 void translate(in short x_translation; in short y_translation);
 void rotate(in float angle_of_rotation); };

class RECTANGLE : GeometryObject
{ extent RECTANGLES)
{ attribute struct Point {short x, short y} Reference_point;
 attribute short Length;
 attribute short Height;
 attribute float Orientation_angle; };

class TRIANGLE : GeometryObject
{ extent TRIANGLES)
{ attribute struct Point {short x, short y} Reference_point;
 attribute short Side_1;
 attribute short Side_2;
 attribute float Side1_side2_angle;
 attribute float Side1_orientation_angle; };

class CIRCLE : GeometryObject
{ extent CIRCLES)
{ attribute struct Point {short x, short y} Reference_point;
 attribute short Radius; };

```

# Object Database Conceptual Design

- Differences between conceptual design of ODB and RDB, handling of:
  - Relationships
  - Inheritance
- Philosophical difference between relational model and object model of data
  - In terms of behavioral specification

# Mapping an EER Schema to an ODB Schema

- Create ODL class for each EER entity type
- Add relationship properties for each binary relationship
- Include appropriate operations for each class
- ODL class that corresponds to a subclass in the EER schema
  - Inherits type and methods of its superclass in ODL schema

# Mapping an EER Schema to an ODB Schema (cont'd.)

- Weak entity types
  - Mapped same as regular entity types
- Categories (union types)
  - Difficult to map to ODL
- An  $n$ -ary relationship with degree  $n > 2$ 
  - Map into a separate class, with appropriate references to each participating class

# The Object Query Language OQL

- Query language proposed for ODMG object model
- Simple OQL queries, database entry points, and iterator variables
  - Syntax: select ... from ... where ... structure
  - Entry point: named persistent object
  - Iterator variable: define whenever a collection is referenced in an OQL query

# Query Results and Path Expressions

- Result of a query
  - Any type that can be expressed in ODMG object model
- OQL orthogonal with respect to specifying path expressions
  - Attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions

# Other Features of OQL

- **Named query**
  - Specify identifier of named query
- OQL query will return collection as its result
  - If user requires that a query only return a single element use `element` operator
- Aggregate operators
- Membership and quantification over a collection

# Other Features of OQL (cont'd.)

- Special operations for ordered collections
- **Group by clause** in OQL
  - Similar to the corresponding clause in SQL
  - Provides explicit reference to the collection of objects within each group or **partition**
- **Having clause**
  - Used to filter partitioned sets

# Overview of the C++ Language Binding in the ODMG Standard

- Specifies how ODL constructs are mapped to C++ constructs
- Uses prefix `d_` for class declarations that deal with database concepts
- Template classes
  - Specified in library binding
  - Overloads operation `new` so that it can be used to create either persistent or transient objects

# Summary

- Overview of concepts utilized in object databases
  - Object identity and identifiers; encapsulation of operations; inheritance; complex structure of objects through nesting of type constructors; and how objects are made persistent
- Description of the ODMG object model and object query language (OQL)
- Overview of the C++ language binding