RNS INSTITUTE OF TECHNOLOGY
Affiliated to VTU, Recognized by GOK, Approved by AICTE, New Delhi
(NAAC 'A+ Grade' Accredited, NBA Accredited (UG - CSE, ECE, ISE, EIE and EEE)
Channasandra, Dr. Vishnuvardhan Road, Bengaluru - 560 098
Ph:(080)28611880,28611881 URL: www.rnsit.ac.in

Estd : 2001

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Semester :V
Subject : Database Management systems
Subject Code : 21CS53

Compiled by:
Mrs. Mamatha Jajur, Mrs. Soumya N G, Mrs. Ancy Thomas
Assistant Professors
Department of CS&E
RNSIT

VISION AND MISSION OF INSTITUTION

Vision

Building RNSIT into a World Class Institution

Mission

To impart high quality education in Engineering, Technology and Management with a Difference, Enabling Students to Excel in their Career by

1. Attracting quality Students and preparing them with a strong foundation in fundamentals so as to achieve distinctions in various walks of life leading to outstanding contributions
2. Imparting value based, need based, choice based and skill based professional education to the aspiring youth and carving them into disciplined, World class Professionals with social responsibility
3. Promoting excellence in Teaching, Research and Consultancy that galvanizes academic consciousness among Faculty and Students
4. Exposing Students to emerging frontiers of knowledge in various domains and make them suitable for Industry, Entrepreneurship, Higher studies, and Research & Development
5. Providing freedom of action and choice for all the Stake holders with better visibility

VISION AND MISSION OF DEPARTMENT

Vision

Preparing Better Computer Professionals for a Real World

Mission

The Department of CSE will make every effort to promote an intellectual and ethical environment by

1. Imparting solid foundations and applied aspects in both Computer Science Theory and Programming practices
2. Providing training and encouraging R&D and Consultancy Services in frontier areas of Computer Science and Engineering with a Global outlook
3. Fostering the highest ideals of ethics, values and creating awareness of the role of Computing in Global Environment
4. Educating and preparing the graduates, highly sought after, productive, and well-respected for their work culture
5. Supporting and inducing lifelong learning

SYLLABUS

Database Management Systems		Semester	5
Course Code	21CS53	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:0:0	SEE Marks	50
Total Hours of Pedagogy	40 hours	Total Marks	100
Credits	03	Exam Hours	03
Examination nature (SEE)	**Theory**		

Course Learning Objectives :

CLO 1. Provide a strong foundation in database concepts, technology, and practice.
CLO 2. Practice SQL programming through a variety of database problems.
CLO 3. Demonstrate the use of concurrency and transactions in database
CLO 4. Design and build database applications for real world problems.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) need not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
6. Introduce Topics in manifold representations.
7. Show the different ways to solve the same problem with different circuits/logic and encourage the students to come up with their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.

MODULE-1 **8 Hours**

Introduction to Databases: Introduction, Characteristics of database approach, Advantages of using the DBMS approach, History of database applications.

Overview of Database Languages and Architectures: Data Models, Schemas, and Instances. Three schema architecture and data independence, database languages, and interfaces, The Database System environment.

Conceptual Data Modelling using Entities and Relationships: Entity types, Entity sets, attributes, roles, and structural constraints, Weak entity types, ER diagrams, Examples

Textbook 1: Ch 1.1 to 1.8, 2.1 to 2.6, 3.1 to 3.7

MODULE-2 **8 Hours**

Relational Model: Relational Model Concepts, Relational Model Constraints and relational database schemas, Update operations, transactions, and dealing with constraint violations.

Relational Algebra: Unary and Binary relational operations, additional relational operations (aggregate, grouping, etc.) Examples of Queries in relational algebra.

Mapping Conceptual Design into a Logical Design: Relational Database Design using ER-to-Relational mapping.

Textbook 1:, Ch 5.1 to 5.3, 8.1 to 8.5, 9.1

8 Hours

MODULE-3

SQL: SQL data definition and data types, specifying constraints in SQL, retrieval queries in SQL, INSERT, DELETE, and UPDATE statements in SQL, Additional features of SQL.

Advances Queries: More complex SQL retrieval queries, Specifying constraints as assertions and action triggers, Views in SQL, Schema change statements in SQL.

Database Application Development: Accessing databases from applications, An introduction to JDBC, JDBC classes and interfaces, SQLJ, Stored procedures, Case study: The internet Bookshop.

Textbook 1: Ch 6.1 to 6.5, 7.1 to 7.4; Textbook 2: 6.1 to 6.6

MODULE-4 8 Hours

Normalization: Database Design Theory: Introduction to Normalization using Functional and Multivalued Dependencies: Informal design guidelines for relation schema, Functional Dependencies, Normal Forms based on Primary Keys, Second and Third Normal Forms, Boyce-Codd Normal Form, Multivalued Dependency and Fourth Normal Form, Join Dependencies and Fifth Normal Form. Examples on normal forms.

Normalization Algorithms: Inference Rules, Equivalence, and Minimal Cover, Properties of Relational Decompositions, Algorithms for Relational Database Schema Design, Nulls, Dangling tuples, and alternate Relational Designs, Further discussion of Multivalued dependencies and 4NF, Other dependencies and Normal Forms

Textbook 1: Ch 14.1 to -14.7, 15.1 to 15.6

MODULE-5 8 Hours

Transaction Processing: Introduction to Transaction Processing, Transaction and System concepts, Desirable properties of Transactions, Characterizing schedules based on recoverability, Characterizing schedules based on Serializability, Transaction support in SQL.

Concurrency Control in Databases: Two-phase locking techniques for Concurrency control, Concurrency control based on Timestamp ordering, Multiversion Concurrency control techniques, Validation Concurrency control techniques, Granularity of Data items and Multiple Granularity Locking.

Textbook 1: Ch 20.1 to 20.6, 21.1 to 21.7

TEXTBOOKS

1. Fundamentals of Database Systems, Ramez Elmasri and Shamkant B. Navathe, 7th Edition, 2017, Pearson.

2. Database management systems, Ramakrishnan, and Gehrke, 3rd Edition, 2014, McGraw Hill

COURSE OUTCOMES: At the end of this course, students are able to:	
CO1	Represent Database with different data modeling concepts.
CO2	Design simple Database Systems.
CO3	Use Structured Query Language (SQL) for building and manipulating Database.
CO4	Develop Application to interact with Databases.
CO5	Analyze and apply normalization for better Database design.
CO6	Demonstrate the use of concurrency control and transaction processing.

CO - PO MATRIX

COURSE OUTCOMES	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	2	-	2	-	-	-	-	-	1	-	-	1	3	1	1	-
CO2	3	2	3	2	-	-	1	-	1	-	-	2	2	1	1	-
CO3	2	1	1	1	3	-	-	-	2	-	-	3	3	3	3	-
CO4	1	-	3	1	3	-	1	1	2	-	-	2	2	3	2	1
CO5	2	2	2	2	-	-	-	-	-	-	-	2	2	1	2	-
CO6	1	-	1	2	2	-	1	1	1	-	-	2	2	2	2	1

Module 1

Introduction to Databases

- Introduction to Database
- Characteristics of database approach
- Advantages of using the DBMS approach
- History of database applications

Overview of Database Languages and Architectures

- Data Models
- Schemas and Instances
- Three schema architecture and data independence
- Database languages and interfaces
- The Database System environment

Conceptual Data Modeling using Entities and Relationships

- Entity types
- Entity sets
- Attributes, roles, and structural constraints
- Weak entity types
- ER diagrams and examples

Textbook 1: Chapter 1.1 to 1.8, 2.1 to 2.6, 3.1 to 3.10

Module 1: Introduction to Databases

Introduction

Databases and database systems are an essential component of life in modern society: most of us encounter several activities every day that involve some interaction with a database.

Databases and database technology have had a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, social media, engineering, medicine, genetics, law, education, and library science.

Database

A **database** is a collection of related data. Or it is an organized collection of related data, stored and accessed electronically.
Data means known facts that can be recorded and that have implicit meaning.
A database has the implicit properties:

- A database represents some aspect of the real world(**miniworld).**
- Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning.
- A database is designed, built, and populated with data for a specific purpose.
- It has an intended group of users and some preconceived applications in which these users are interested.

A database has some source from which data is derived, some degree of interaction with events in the real world, and an audience that is actively interested in its contents. It can be of any size and complexity. A database may be generated and maintained manually or it may be computerized.

Database management system (DBMS)

A database management system (DBMS) is a computerized system that enables users to create and maintain a database.
Definition: "The DBMS is a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, *manipulating*, and *sharing* databases among various users and applications".

Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database.
Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS.
Manipulating a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.
Sharing a database allows multiple users and programs to access the database simultaneously.

- The database definition or descriptive information is also stored by the DBMS in the form of a **database catalog or dictionary**; it is called **meta-data**.
- An **application program** accesses the database by sending queries or requests for data to the DBMS.
- A **query** typically causes some data to be retrieved.
- A **transaction** may cause some data to be read and some data to be written into the database.
- The DBMS also *protects* the database and *maintaining* it over a long period of time.
- **Protection** includes *system protection* against hardware or software malfunction (or crashes) and *security protection* against unauthorized or malicious access.

- ○ A typical large database may have a lifecycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.
- ○ In the database approach, a single repository maintains data that is defined once and then accessed by various users repeatedly through queries, transactions, and application programs.
- ○ The database and DBMS software together form a **database system**.

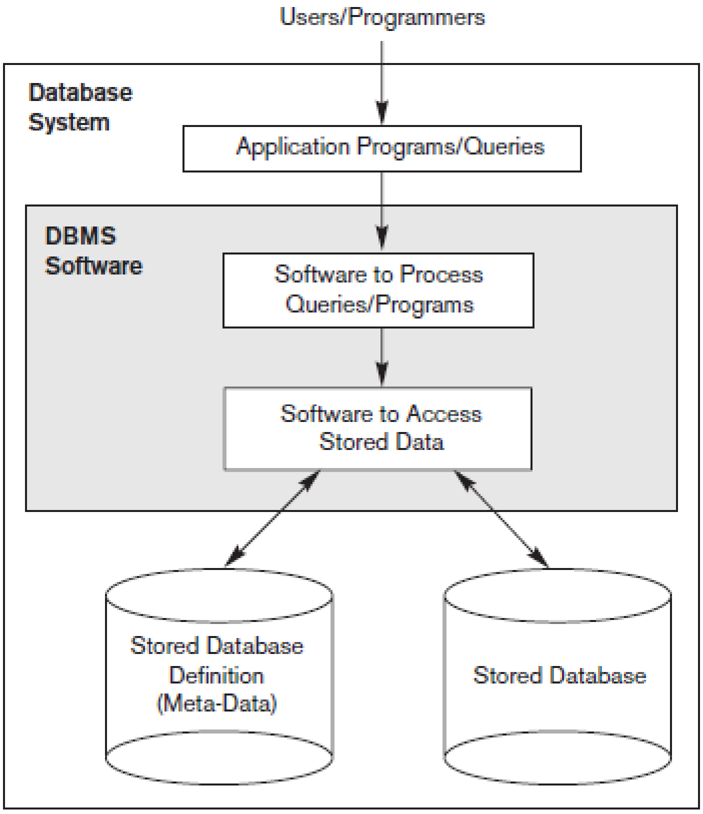
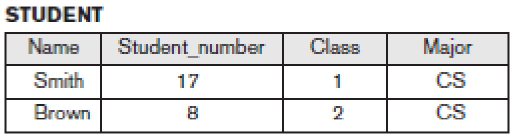

Users/Programmers
Database System
Application Programs/Queries
DBMS Software
Software to Process Queries/Programs
Software to Access Stored Data
Stored Database Definition (Meta-Data)
Stored Database

Fig 1. A simplified database system environment

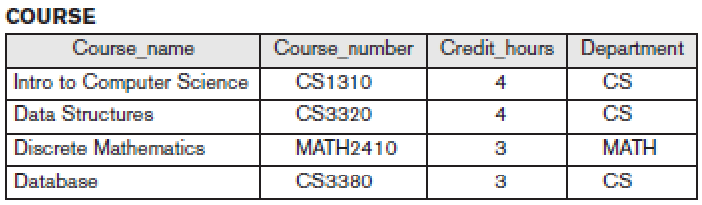
An Example

UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment.



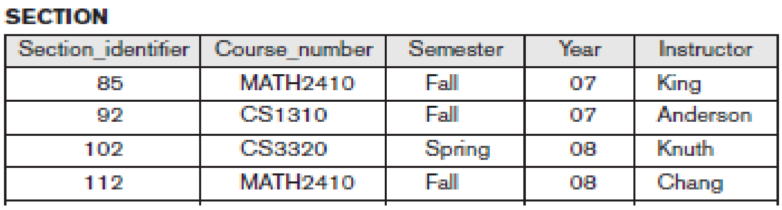
STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS



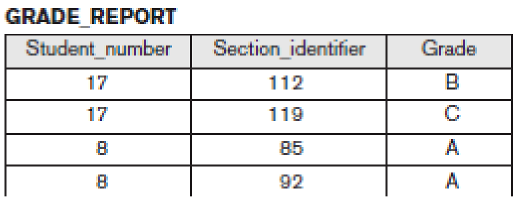
COURSE

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



SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang



GRADE_REPORT

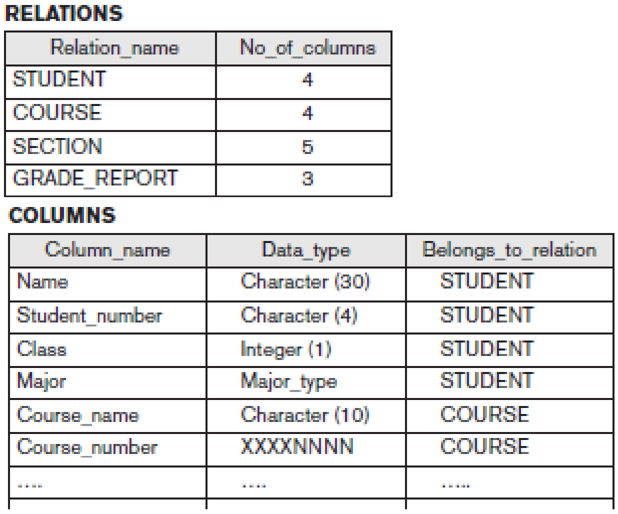
Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A

Fig 2: University Database

Characteristics of the Database Approach

■ Self-describing nature of a database system

The database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the DBMS catalog, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data**, and it describes the structure of the primary database.



RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3

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
.....	.....	.....

An example of a database catalog for the UNIVERSITY database.

■ Insulation between programs and data, and data abstraction

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require *changing all programs* that access that file.

By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. This property is called **program-data independence**.

In database systems, users can define operations on data as part of the database definitions. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence**.

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a **conceptual representation** of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation.

■ Support of multiple views of the data

A database typically has many types of users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views.

■ Sharing of data and multiuser transaction processing

A multiuser DBMS must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct.

For example, when several reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger.

A **transaction** is an *executing program* or *process* that includes one or more database accesses, such as reading or updating of database records. Each transaction is supposed to execute a logically correct database access if executed in its entirety without interference from other transactions. The DBMS must enforce several transaction properties.

The **isolation** property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently.

The **atomicity** property ensures that either all the database operations in a transaction are executed or none are.

File System vs DBMS – Difference between File System and DBMS

File Management System	**Database Management System**
File System is a general, easy-to-use system to store general files which require less security and constraints.	Database management system is used when security constraints are high.
Data Redundancy is more in file management system.	Data Redundancy is less in database management system.
Data Inconsistency is more in file system.	Data Inconsistency is less in DBMS
Centralization is hard	Centralization is achieved
User locates the physical address of the files to access data in File Management System.	User is unaware of physical address data.
Security is low	Security is high
Stores unstructured data as isolated data files/entities.	Stores structured data which have well defined constraints and interrelation.

Actors on the Scene

Database Administrators

Database Administrators responsible for

- Administering primary resource database itself, and the secondary resource DBMS and related software.
- Authorizing access to the database, coordinating and monitoring its use, and acquiring software and hardware resources as needed.
- The DBA is accountable for problems such as security breaches and poor system response time.

Database designers

Database designers are responsible for

- Identifying the data to be stored in the database and for choosing appropriate structures to represent and store the database.
- To communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements.
- To interact with each potential group of users and develop **views** of the database that meet the data and processing requirements of these groups.
- Each view is then analyzed and *integrated* with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

End Users

End users are the people, whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

■ **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query interface to specify their requests and are typically middle- or high-level managers or other occasional browsers. They learn only a few facilities that they may use repeatedly.

■ **Naive** or **parametric end users** make up a sizable portion of database end users. They constantly querying and updating the database, using standard types of queries and update called **canned transactions**—that have been carefully programmed and tested. They need to learn very little about the facilities provided by the DBMS.

Example: Bank customers and tellers check account balances and post withdrawals and deposits.

■ **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements. They try to learn most of the DBMS facilities.

■ **Standalone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. They typically become very proficient in using a specific software package.

Example is the user of a financial software package that stores a variety of personal financial data.

System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for standard canned transactions that meet these requirements.

Application programmers implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such software engineer should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

Workers behind the Scene

These persons are typically not interested in the database itself. They are associated with the design, development and operation and maintenance of DBMS software and system environment. They are called as workers behind the scene.

Includes: DBMS Designers and Implementers, Tool Developers (Packages for database design, performance monitoring, natural language, or GUI, prototyping, simulation, and test data generation), Operators and maintenance personnel.

Advantages of Using the DBMS Approach

- Controlled redundancy
- Restricted unauthorized access (Security at various levels)
- Providing Persistent Storage for Program Objects
- Providing Storage Structures and Search Techniques for Efficient Query Processing
- Providing multiple users and interfaces
- Representing complex relationships among data
- Enforcing integrity constraints
- Providing backup and recovery
- Representing Complex Relationships among Data
- Handling large volume of data and allowing multiple views

1. Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example of Figure 2; here, two groups of users might be the course registration personnel and the accounting office.

In the traditional approach, each group independently keeps files on students. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Much of the data is stored twice: once in the files of each user group. This **redundancy** in storing the same data multiple times leads to several problems:-

First, there is the need to perform a single logical update-such as entering data on a new student-multiple times: once for each file where student data is recorded. This leads to *duplication of effort.*

Second, *storage space* is *wasted* when the same data is stored repeatedly, and this problem may be serious for large databases.

Third, files that represent the same data may become *inconsistent.* This may happen because an update is applied to some of the files but not to others. Even if an update-such as adding a new student-is applied to all the appropriate files, the data concerning the student may still be *inconsistent* because the updates are applied independently by each user group.

For example, one user group may enter a student's birthdate erroneously as JAN-19-1984, whereas the other user groups may enter the correct value of JAN-29-1984.

In the database approach, the views of different user groups are integrated during database design. Ideally, In database, all the data items such as student's name or birth date are stored in *only one place (single repository)*. This ensures consistency, and it saves storage space.

2. Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database.

For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to retrieve and to update.

Hence, the type of access operation-retrieval or update-must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database.

A DBMS provides a **security and authorization subsystem**, which the DBA uses to create accounts and to specify account restrictions. The DBMS should then enforce these restrictions automatically.

3. Providing Persistent Storage for Program Objects

Databases can be used to provide persistent storage for program objects and data structures. Programming languages typically have complex data structures, such as record types in Pascal or class definitions in C++ or Java. The values of program variables are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable structure.

Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be **persistent,** since it survives the termination of program execution and can later be directly retrieved by another c+ + program. The persistent storage of program objects and data structures is an important

4. Providing Storage Structures for Efficient Query Processing

Database systems must provide capabilities for *efficiently executing queries and updates.* Because the database is typically stored on disk, the DBMS must provide specialized data structures to speed up disk search for the desired records.

Auxiliary files called **indexes** are used for this purpose. Indexes are typically based on tree data structures or hash data structures, suitably modified for disk search.

In order to process the database records needed by a particular query, those records must be copied from disk to memory. Hence, the DBMS often has a **buffering** module that maintains parts of the database in main memory buffers.

The **query processing and optimization** module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures. The choice of which indexes to create and maintain is part of *physical database* design *and tuning*, which is one of the responsibilities of the DBA staff.

5. Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery.

For example, if the computer system fails in the middle of a complex update transaction, the **recovery subsystem** is responsible for making sure that the database is restored to the state it was in before the transaction started executing.

Alternatively, the recovery subsystem could ensure that the transaction is resumed from the point at which it was interrupted so that its full effect is recorded in the database.

6. Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces.

These include query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for stand-alone users.

Both forms-style interfaces and menu-driven interfaces are commonly known as graphical user interfaces (GU Is).

Many specialized languages and environments exist for specifying GUls. Capabilities for providing Web GUl interfaces to a database- or Web-enabling a database-are also quite common.

7. Representing Complex Relationships among Data

A database may include numerous varieties of data that are interrelated in many ways. Consider the example shown in Figure 2. The record for Brown in the STUDENT file is related to four records in the GRADE_REPDRT file.

Similarly, each section record is related to one course record as well as to a number of GRADE_REPDRT records-one for each student who completed that section.

A DBMS have the capability to represent a variety of complex relationships among the data as well as to retrieve and update related data easily and efficiently.

8. Enforcing Integrity Constraints

Most database applications have certain integrity constraints that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints.

The simplest type of integrity constraint involves specifying a data type for each data item.

For example, in Figure 2, it may be specified that the value of the Class data item within each STUDENT record must be an integer between 1 and 5 and that the value of Name must be a string of no more than 30 alphabetic characters.

A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files.

For example, in Figure 2, it can be specified that "every section record must be related to a course record."

Another type of constraint specifies uniqueness on data item values, such as "every course record must have a unique value for CourseNumber." These constraints are derived from the meaning or semantics of the data and of the miniworld it represents.

It is the database designers' responsibility to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry.

A data item may be entered erroneously and still satisfy the specified integrity constraints. For example, if a student receives a grade of A but a grade of C is entered in the database, the DBMS *cannot* discover this error automatically, because C is a valid value for the Grade data type. Such data entry errors can only be discovered manually (when the student receives the grade and complains) and corrected later by updating the database. However, a grade of Z can be rejected automatically by the DBMS, because Z is not a valid value for the Grade data type.

9. Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for inferencing new information from the stored database facts. Such systems are called **deductive database systems**.

For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as rules, which when compiled and maintained by the DBMS can determine all students on probation.

In a traditional DBMS, an explicit *procedural program code* would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs.

10. Additional Implications of Using the Database Approach

This section discusses some additional implications of using the database approach that can benefit most organizations: -

Potential for Enforcing Standards: The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization.

Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on.

The DBA can enforce standard in a centralized database environment more easily than in an environment where each user group has control of its own files and software.

Reduced Application Development Time: A prime selling feature of the database approach is that developing a new application-such as the retrieval of certain data from the database for printing a new report-takes very little time. Designing and implementing a new database from scratch may take more time than writing a single specialized file application.

However, once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities.

Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system.

Flexibility: It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs information not currently in the database.

In response, it may be necessary to add a file to the database or to extend the data elements in an existing file.

Availability of Up-to-Date Information: A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update.

This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

Economies of Scale: The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data processing personnel in different projects or departments.

This enables the whole organization to invest in more powerful processors, storage devices, or communication gear, rather than having each department purchase its own (weaker) equipment. This reduces overall costs of operation and management.

A Brief History of Database Applications

1. Relational Model: proposed in 1970 by E.F. Codd (IBM), first commercial system in 1981-82. Now in several commercial products (DB2, ORACLE, SQL Server, SYBASE, INFORMIX).
2. Network Model: the first one to be implemented by Honeywell in 1964-65 (IDS System). Adopted heavily due to the support by CODASYL (CODASYL - DBTG report of 1971). Later implemented in a large variety of systems - IDMS (Cullinet - now CA), DMS 1100 (Unisys), IMAGE (H.P.), VAX -DBMS (Digital Equipment Corp.).
3. Hierarchical Data Model: implemented in a joint effort by IBM and North American Rockwell around 1965. Resulted in the IMS family of systems. The most popular model. Other system based on this model: System 2k (SAS inc.)
4. Object-oriented Data Model(s): several models have been proposed for implementing in a database system. One set comprises models of persistent O-O Programming Languages such as C++ (e.g., in OBJECTSTORE or VERSANT), and Smalltalk (e.g., in GEMSTONE). Additionally, systems like $O_{2,}$ ORION (at MCC - then ITASCA), IRIS (at H.P.- used in Open OODB).
5. Object-Relational Models: Most Recent Trend. Started with Informix Universal Server. Exemplified in the latest versions of Oracle-10i, DB2, and SQL Server etc. systems.

Hierarchical Model

ADVANTAGES:

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

DISADVANTAGES:

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

Network Model

<u>ADVANTAGES:</u>

- 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.

<u>DISADVANTAGES:</u>

- 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 Database Systems

- 1950s and early 1960s:
  - Data processing using magnetic tapes for storage
    - Tapes provide only sequential access
  - Punched cards for input
- Late 1960s and 1970s:
  - Hard disks allow direct access to data
  - Network and hierarchical data models in widespread use
  - Ted Codd defines the relational data model
    - Would win the ACM Turing Award for this work
    - IBM Research begins System R prototype
    - UC Berkeley begins Ingres prototype
  - High-performance (for the era) transaction processing
- 1980s:
  - Research relational prototypes evolve into commercial systems
    - SQL becomes industry standard
  - Parallel and distributed database systems
  - Object-oriented database systems
- 1990s:
  - Large decision support and data-mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce
- 2000s:
  - XML and XQuery standards
  - Automated database administration
  - Increasing use of highly parallel database systems
  - Web-scale distributed data storage systems

When not to use a DBMS [Limitations]

Main costs of using a DBMS:

- High initial investment in hardware, software, training and possible need for additional hardware.
- Overhead for providing generality, security, recovery, integrity, and concurrency control.
- Generality that a DBMS provides for defining and processing data.

When a DBMS may be unnecessary:

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

Application of Database System

- Railway Reservation System
- Library Management System
- Banking
- Universities and colleges
- Credit card transactions
- Social Media Sites
- Telecommunications
- Finance
- Military
- Online Shopping
- Human Resource Management
- Manufacturing
- Airline Reservation system

Overview of Database Languages and Architectures

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with a client /server system architecture.

Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. **Data abstraction** generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. So that different users can perceive data at their preferred level of detail.

A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve data abstraction. By *structure of a database* means the data types, relationships, and constraints that apply to the data.
Data models also include a set of **basic operations** for specifying retrievals and updates on the database.

Categories of Data Models

- **High-level** or **conceptual data models**
- **Representational** or **implementation data models**
- **Low-level** or **physical data models**

High-level or conceptual data models provide concepts that are close to the way many users perceive data. It uses concepts such as entities, attributes, and relationships.

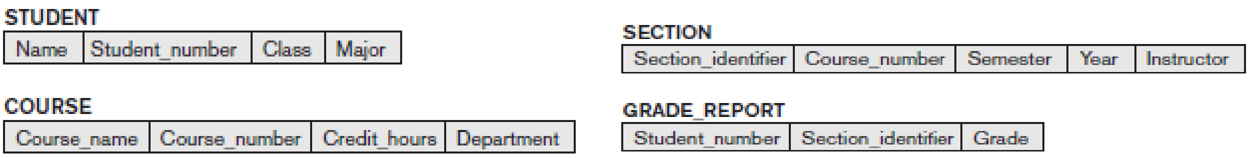
Example an **Entity–relationship model.** An **entity** represents a real-world object or concept, such as an employee or project. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project.

Representational or implementation data models Provide concepts that fall between the above two, used by many commercial DBMS implementations. Used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, **network** and **hierarchical models**—that have been widely used in the past. Here data is represented by using record structures and hence are sometimes called **record-based data models**.

Low-level or physical data models provide concepts that describe the details of how data is stored on the computer storage media. Concepts provided are generally meant for computer specialists, not for end users. Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths.

Schemas, Database State or Instances

- **Database Schema:** The description of a database that includes descriptions of the database structure, data types, and the constraints on the database.
- **Schema Diagram:** A displayed schema is called a **schema diagram**. A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints.



STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

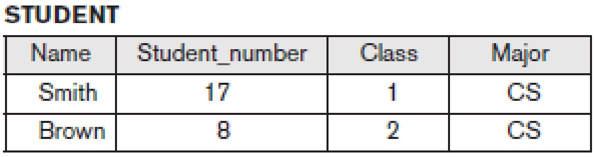
SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

Schema diagram for the UNIVERSITY database



STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

Example of a database state for STUDENT

- **Schema construct:** Each object in the schema—such as STUDENT or COURSE is called a schema construct.
- **Database state or snapshot**: The data in the database at a particular moment in time is called a database state or snapshot. It is also called the current set of **occurrences** or **instances** in the database. At any point in time, the database has a current state. The DBMS is partly responsible for ensuring that every state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema.
- The database schema changes very infrequently. The database state changes every time the database is updated.
- The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

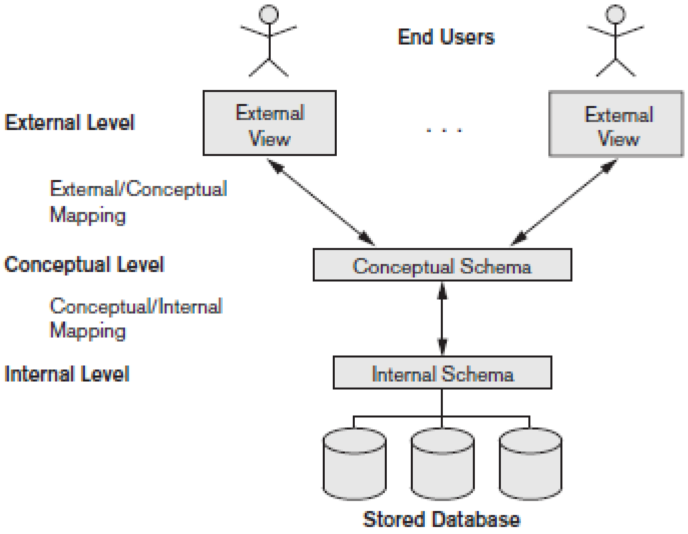
Three-Schema Architecture

The goal of the three-schema architecture is to separate the user applications from the physical database. Here schemas can be defined at the following three levels:

- **Internal level**
- **Conceptual level**
- **External** or **view level**

The **internal level** has an **internal schema**, which describes the physical storage structure of the database. It uses a physical data model and describes the complete details of data storage and access paths for the database.
The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. It hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema.

The **external** or **view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. Typically implemented using a representational data model.



The three-schema architecture

- The user can easily visualize the schema levels in a database system.
- Most DBMSs do not separate the three levels completely and explicitly.
- Here, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database.
- If the request is database retrieval, the data extracted from the stored database must be reformatted to match the user's external view.
- The processes of transforming requests and results between levels are called **mappings**.

Data Independence

Data independence is the capacity to change the schema at one level of a database system without having to change the schema at the next higher level.
Two types of data independence:
Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs. The conceptual schema may be changed to expand the database, to change constraints, or to reduce the database.
After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before.

Physical data independence is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update.
If the same data as before remains in the database, we should not have to change the conceptual schema.

DBMS Languages

A DBMS has appropriate languages and interfaces to express database queries and updates. Database languages can be used to read, store and update the data in the database.

Types of DBMS languages:

Data Definition Language (DDL)
Data Manipulation Language (DML)
Data Control Language (DCL)
Transaction Control Language (TCL)
Storage definition language (SDL)
View definition language (VDL)

Data Definition Language (DDL)

- DDL is used to define the database schema.
- It is used to create schema, tables, indexes, constraints etc. in the database.
- Using the DDL statements, you can create the skeleton of the database.
- The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

Here are some tasks that come under DDL:

- **Create**: It is used to create the database or its objects (like table, index, function, views, store procedure and triggers).

- **Alter**: It is used to alter the structure of the database.
- **Drop**: It is used to delete objects from the database.
- **Truncate**: It is used to remove all records from a table, including all spaces allocated for the records are removed.
- **Rename**: It is used to rename an object.
- **Comment**: It is used to comment to the data dictionary.

These commands are used to update the database schema that's why they come under Data definition language.

Data Manipulation Language (DML)

DML is used for accessing and manipulating data in a database. Like retrieval, insertion, deletion, and modification of the data. It handles user requests. There are two main types of DMLs.

- <u>A high-level or nonprocedural DML</u> statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. Also called **set-at-a-time** or **set-oriented** DMLs.
- <u>A low-level or procedural DML</u> must be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Also called as **record-at-a-time** DMLs.
- Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**. On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. In

Here are some tasks that come under DML:

- **Select**: It is used to retrieve data from a database.
- **Insert**: It is used to insert data into a table.
- **Update**: It is used to update existing data within a table.
- **Delete**: It is used to delete all records from a table.

Data Control Language (DCL)

- DCL mainly deals with the rights, permissions and other controls of the database system.

Here are some tasks that come under DCL:

- **Grant**: It is used to give user access privileges to a database.
- **Revoke**: It is used to take back permissions from the user.

There are the following operations which has authorization of Revoke: CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

Transaction Control Language (TCL)

TCL is used to run the changes made by the DML statement. TCL commands deals with the transaction within the database. Here are some tasks that come under TCL:

- **Commit**: It is used to save the transaction on the database.
- **Rollback**: It is used to restore the database to original since the last Commit.
- **SAVEPOINT**–sets a savepoint within a transaction.
- **SET TRANSACTION**–specify characteristics for the transaction.

Storage definition language (SDL)

Storage definition language (SDL) is used to specify the internal schema. In most relational DBMSs today, there *is no specific language* that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage of files.

View definition language (VDL) View definition language (VDL), to specify user views and their mappings to the conceptual schema, but in most DBMSs *the DDL is used to define both conceptual and external schemas*. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries

DBMS Interfaces

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

Menu-based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options (called **menus)** that lead the user through the formulation of a request.

Apps for Mobile Devices. These interfaces present mobile users with access to their data. For example, banking, reservations, and insurance companies.

Forms-based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the **form** entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Example: SQL*Forms, Oracle Forms

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

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to *understand* them. It usually has its own *schema*, as well as a dictionary of important words.

Keyword-based Database Search. It accepts strings of natural language (like English or Spanish) words and match them with documents at specific sites (for local search engines) or Web pages on the Web at large (for engines like Google or Ask).

Speech Input and Output The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place. Applications with limited vocabularies, such as inquiries for telephone directory, flight arrival/departure.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactionssuch as account deposits or withdrawals, or balance inquiries.

Interfaces for the DBA. Used only by the DBA staff to execute privileged commands. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

DBMS Programming Language Interfaces

Programmer interfaces for embedding DML in programming languages:

1. Embedded Approach: e.g embedded SQL (for C, C++, etc.), SQLJ (for Java)

2. Procedure Call Approach: e.g. JDBC for Java, ODBC for other programming languages

3. 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

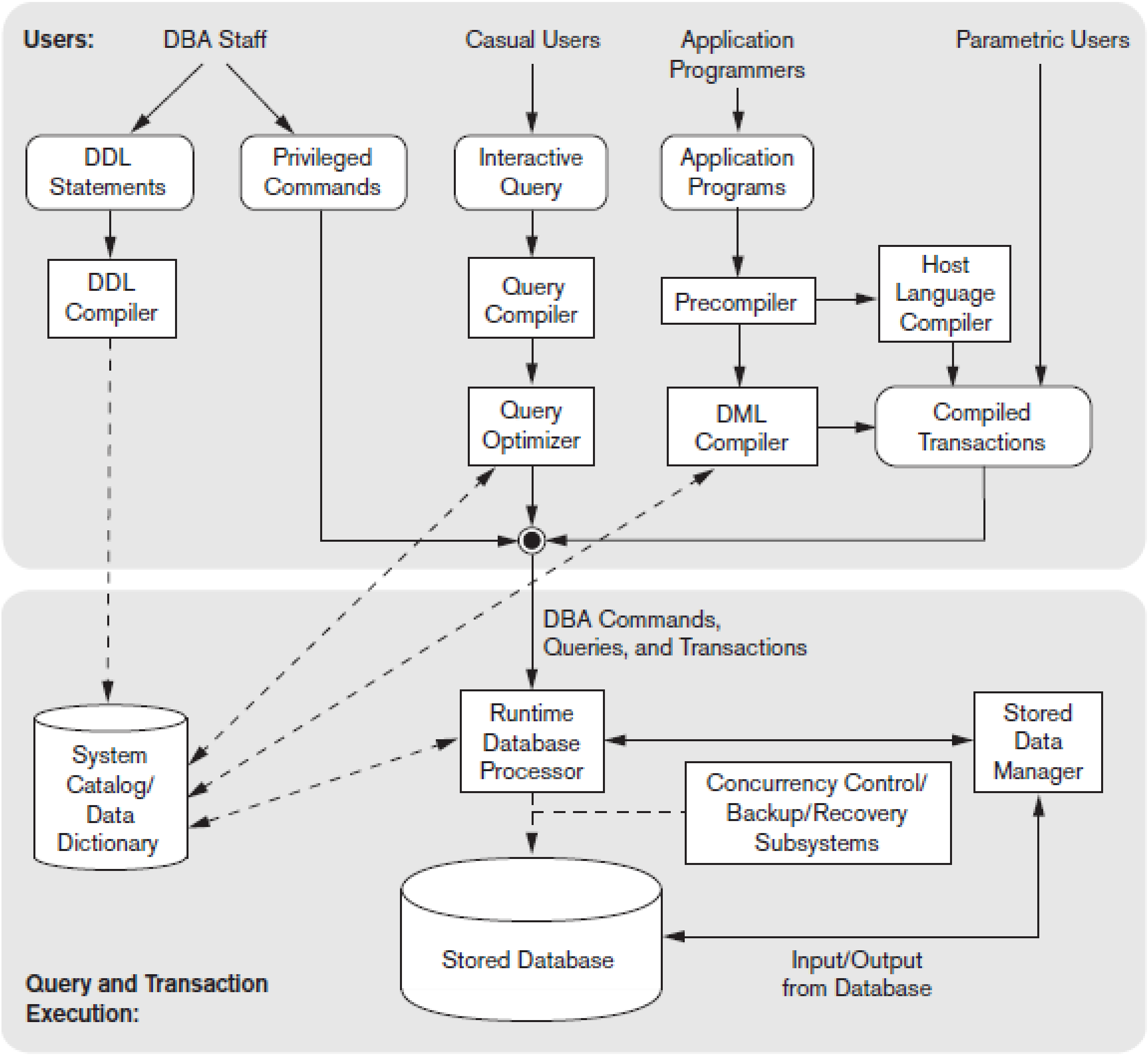
The Database System Environment

A DBMS is a complex software system. The typical DBMS components are divided into two parts:
The various users of the database environment and their interfaces.
The internal modules of the DBMS responsible for storage of data and processing of transactions.

- The database and the DBMS catalog are usually stored on disk.
- Access to the disk is controlled primarily by the **operating system** (**OS**), which schedules disk read/write.
- A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

Database Users

- **DBA staff:** The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands. The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog.
- **Casual users** and persons with occasional need for information from the database interact using the interactive query interface or to access canned transactions.These queries are parsed and validated for correctness of the query syntax by a **query compiler** that compiles them into an internal form and then is subjected to query optimization. The **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of efficient search algorithms during execution.
- **Application programmers** write programs in host languages such as Java, C, or C++ that are submitted to a precompiler, which extracts DML commands from an application program. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler.
- **Parametric users** who do data entry work by supplying parameters to predefined transactions.



Component modules of a DBMS and their interactions

1. The **catalog** includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints.In addition, the catalog stores many other types of information that are needed by the DBMS modules, which can then look up the catalog information as needed.

- **Runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the **system catalog** and may update it with statistics.
- Runtime database processor also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory.
- The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory.
- **concurrency control** and **backup and recovery systems modules** are used for transaction management.

Database System Utilities

Most DBMSs have **database utilities** that help the DBA manage the database system.

- **Loading**: A loading utility is used to load existing data files—such as text files or sequential files—into the database.
- **Backup**. A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium.
- **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations and create new access paths to improve performance.
- **Performance monitoring.** Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.
- Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

Other Tools

Data dictionary / repository:

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

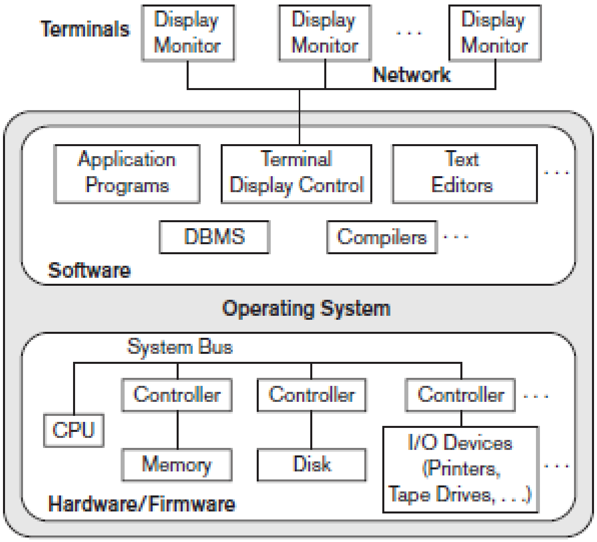
Application Development Environments and CASE (computer-aided software engineering) tools: Examples:

- PowerBuilder (Sybase)
- JBuilder (Borland)
- JDeveloper 10G (Oracle)

Centralized and Client/Server Architectures for DBMSs

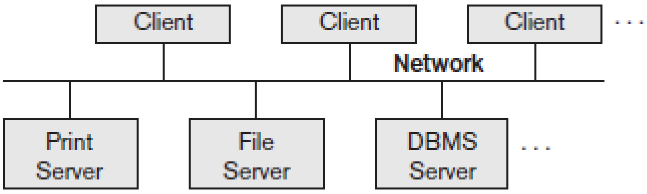
Centralized DBMSs Architecture

2. Older architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality.
3. Most users accessed the DBMS via computer terminals that did not have processing power and only provided display capabilities.
4. Therefore, all processing was performed remotely on the computer system housing the DBMS, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.



Basic Client/Server Architectures

- The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network.
- The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines.
- The resources provided by specialized servers can be accessed by many client machines.
- The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications.
- A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access.

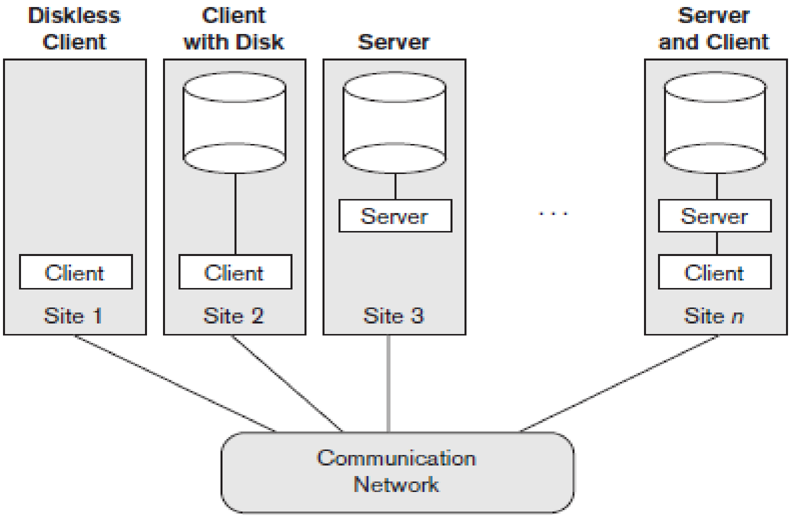


Logical two-tier client/server architecture.

Two-Tier Client/Server Architectures for DBMSs

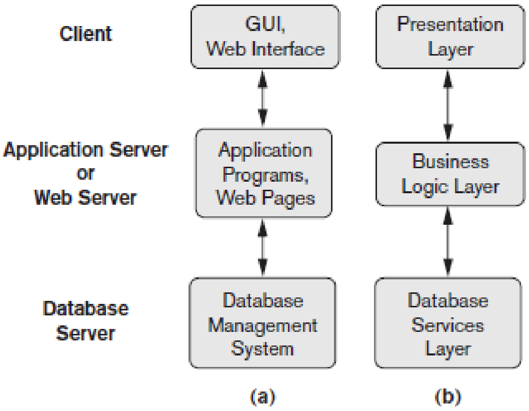
- In the two-tier architectures, the software components are distributed over two systems: client and server.
- The advantages are its simplicity and seamless compatibility with existing systems.
- The user interface programs and application programs can run on the client side.
- When DBMS access is required, the program establishes a connection to the DBMS, the client program can communicate with the DBMS.
- A standard called Open Database Connectivity (ODBC) provides an application programming interface (API), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed.

- A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites.
- Any query results are sent back to the client program, which can process and display the results as needed.



Three-Tier and n-Tier Architectures for Web Applications

- The three-tier architecture, which adds an intermediate layer between the client and the database server.
- This intermediate layer or middle tier is called the application server or the Web server, runs application programs and storing business rules that are used to access data from the database server.
- Improves database security by checking a client's credentials before forwarding a request to the database server.
- Clients contain user interfaces and Web browsers.
- The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to the users.
- Thus, the user interface, application rules, and data access act as the three tiers.



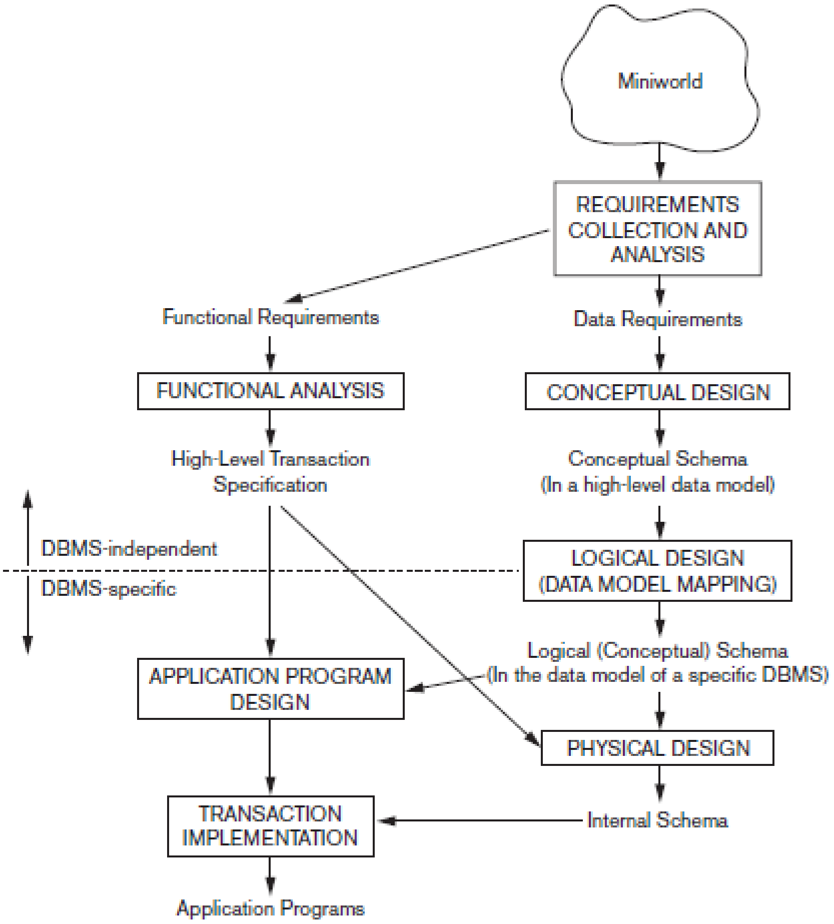
Logical three-tier client/server architecture

Conceptual Data Modeling using Entities and Relationships

Conceptual modeling is a very important phase in designing a successful database application.

Using High-Level Conceptual Data Models for Database Design

- The database design process begins with **requirements collection and analysis**.
- During this step, the database designers interview prospective database users to understand and document their **data requirements**.
- In parallel with this, specify the known **functional requirements** of the application,like user defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates.
- The next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**.


Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Data Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

- The **conceptual schema** is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints;
- During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis and confirm all the identified functional requirements are meet.
- Modifications to the conceptual schema can be introduced if required.
- The next step is the actual implementation of the database, using a commercial DBMS, using an **implementation data model**—such as the relational (SQL) model—so the conceptual schema is transformed from the high-level data model into the implementation data model, resulting in a database schema.
- This step is called **logical design** or **data model mapping**
- Data model mapping is often automated or semi-automated within the database design tools.
- The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified.

- In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications.

The ER model Concepts

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

Entities and Attributes

Entity: The basic concept that the ER model represents is an entity, which is a thing or object in the real world with an independent existence.
An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).
Attributes: Each entity has attributes. The attributes are particular properties that describe entity. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.
The attribute values that describe each entity become a major part of the data stored in the database.

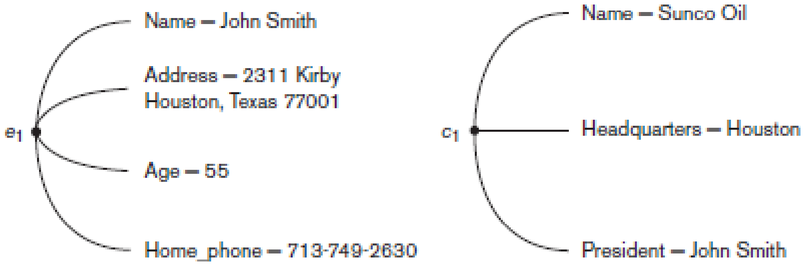


Figure 3.3
Two entities, EMPLOYEE e_1, and COMPANY c_1, and their attributes.

<u>**Types of attributes**</u>

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings.
Example, the Address attribute of the EMPLOYEE entity can be subdivided into Street_address, City, State, and Zip,3 with the values '2311 Kirby', 'Houston', 'Texas', and '77001'.

Simple (Atomic) Attributes that are not divisible are called simple or atomic attributes.
Example: the age of the EMPLOYEE entity.

Single-Valued attributes have a single value for a particular entity.
Example: Age is a single-valued attribute of a person.

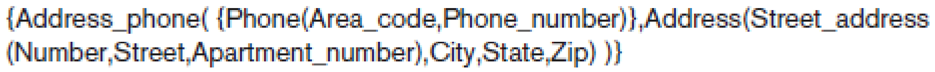
Multivalued Attributes can have a set of values for the same entity.
Example: Colors attribute for a car, or a College degrees attribute for a person.

A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity.

Stored and Derived Attributes. In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current date and the value of that person's Birth_date.
The Age attribute is hence called a **derived** attribute and is said to be derivable from the Birth_date attribute, which is called a **stored** attribute.

Some attribute values can be derived from related entities; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that
department.

Complex Attributes. The composite and multivalued attributes can be nested arbitrarily. By grouping components of a composite attribute between parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces { }. Such attributes are called **complex** attributes. Example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified. Both Phone and Address are themselves composite attributes.



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

NULL Values. A special value called NULL is used for situations where attribute **is not applicable or unknown.**

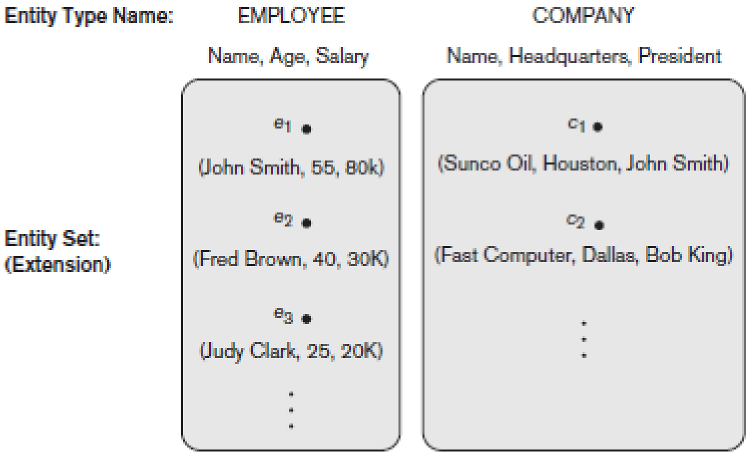
NULL can be used if

1. A particular entity **may not have an applicable value** for an attribute.
   Example,Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes.
   College_degrees attribute applies only to people with college degrees.
2. We **do not know the value of an attribute** for a particular entity, example, if we do not know the home phone number of 'John Smith'. The unknown category of NULL can be further classified into two cases.
   - when it is **known that the attribute value exists but is missing**—for instance, if the Height attribute of a person is listed as NULL.
   - when it is **not known whether the attribute value exists**—for example, if the Home_phone attribute of a person is NULL.

Entity Types, Entity Sets, Keys, and Value Sets

Entity Types: An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each Entity type in the database is described by its name and attributes.
A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees and these employee entities share the same attributes, but each entity has its *own value*(*s*) for each attribute. Figure shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for each.
An entity type is represented in ER diagrams as a **rectangular box** enclosing the entity type name.

Entity Sets: The collection of all entities of a particular entity type in the database at any point in time is called an **entity set** or **entity collection**; the entity set is usually referred to using the same name as the entity type.
For example, EMPLOYEE refers to both a *type of entity* as well as the current collection *of all employee entities* in the database.
An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure.
The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.



Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

Key Attributes of an Entity Type

An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely.
Example: For the STUDENT entity type, a typical key attribute is USN.

- Sometimes several attributes together form a key, that is *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, then define a *composite attribute* and designate it as a key attribute of the entity type.
- Notice that such a composite key must be *minimal*;
- Superfluous attributes must not be included in a key.
- In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval.
- Uniqueness property must hold for *every entity set* of the entity type.
- This key constraint is derived from the constraints of the miniworld that the database represents.

Relationship Types, Relationship Sets, Roles, and Structural Constraints

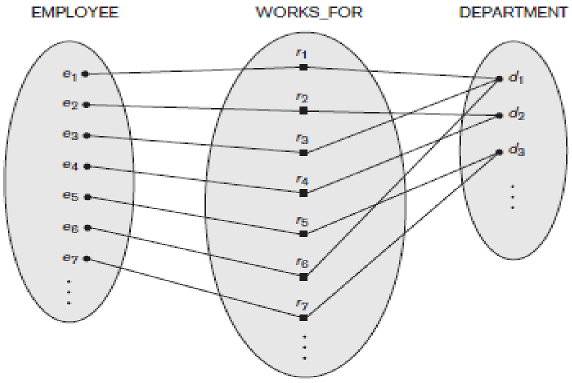
There are several implicit relationships among the various entity types.In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department; the attribute Controlling_department of PROJECT refers to the department that controls the project;
In the ER model, these references should not be represented as attributes but as relationships.

Relationship Types

A **relationship type R** among n entity types E1, E2, . . . , En defines a set of associations—or a relationship set—among entities from these entity types.

The **relationship set R** is a set of relationship instances ri, where each ri associates n individual entities (e1, e2, . . . , en), and each entity ej in ri is a member of entity set Ej, 1 ≤ j ≤ n.

- Informally, each relationship instance ri in R is an association of entities, where the association includes exactly one entity from each participating entity type.
- For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works.
- Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity.
- In ER diagrams, relationship types are displayed as **diamond-shaped boxes**, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box



Degree of a Relationship Type

The degree of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two as shown in above figure.

A relationship type of degree two is called **binary**, and one of degree three is called **ternary**.

Relationships can generally be of any degree, but the ones most common are binary relationships.

Relationships as Attributes

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

Role Names

Each entity type that participates in a relationship type plays a particular role in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and it helps to explain what the relationship means.

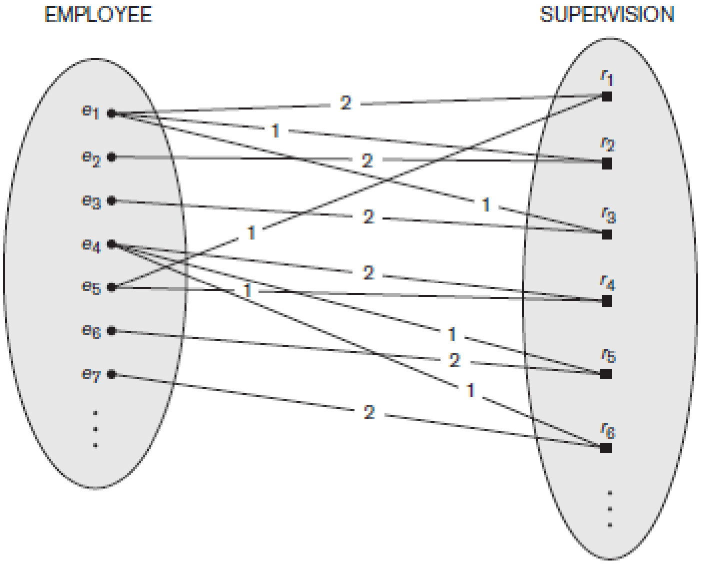
Example: In the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Recursive Relationships

In some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for distinguishing the meaning of the role that each participating entity plays. Such relationship types are called **recursive relationships** or **self-referencing relationships**.

Example: The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set.

Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or *boss*), and once in the role of *supervisee* (or *subordinate*).



Constraints on Binary Relationship Types

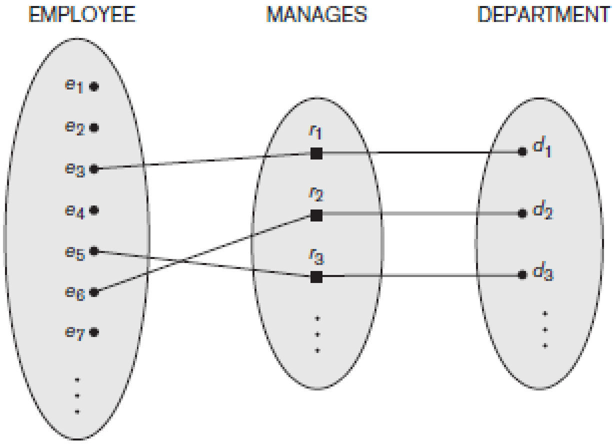
- Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set.
- These constraints are determined from the miniworld situation that the relationships represent. Example: if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema.
- Two main types of binary relationship constraints: ***cardinality ratio* and *participation*.**
- The cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

Cardinality Ratios for Binary Relationships:

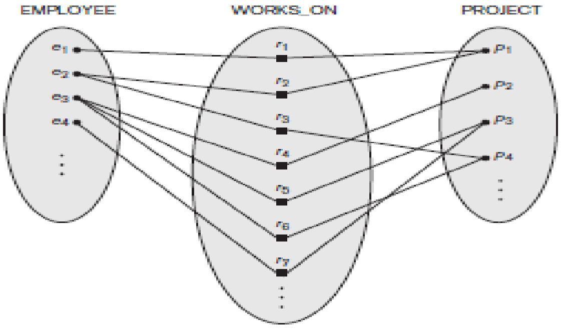
The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate in. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

Example 1: In the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, that is each department can have any number of employees (N), but an employee can work for at most one department (1).

Example 2: binary relationship MANAGES is of cardinality ratio 1:1, an employee can manage atmost one department and a department can have at most one manager.



Example 3: The relationship type WORKS_ON is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.



Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds. Also, *maximum number* on participation can be specified, such as 4 or 5.

Participation Constraints and Existence Dependencies

The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. It specifies the *minimum* number of relationship instances that each entity can participate in and thus is also called the **minimum cardinality constraint**.

There are two types of participation constraints

> **Total participation constraints:** If every instance of entity type E participate in at least one relationship instance r, the participation is said to be total. Total participation is also called **existence dependency**.
> Example: E*very* employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance. Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation**, meaning that every entity in *the total set* of employee entities must be related to a department entity via WORKS_FOR.
>
> **Partial participation constraints:** If only some of the instances of the entity type E participate in the relationship, the participation is said to be partial.

Example: Every employee is not expected to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial**, meaning that *some* or *part of the set of* employee entities are related to some department entity via MANAGES, but not necessarily all.

In ER diagrams, total participation is displayed as a ***double line*** connecting the participating entity type to the relationship, whereas partial participation is represented by a ***single line.***

Weak Entity Types

- Entity types that do not have key attributes of their own are called **weak entity types**.
- Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values.
- This other entity type is called the **identifying** or **owner entity type**, and the relationship type that relates a weak entity type to its owner is called the **identifying relationship** of the weak entity type.
- A weak entity type always has a *total participation constraint* with respect to its identifying relationship because a weak entity cannot be identified without an owner entity.
- A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are *related to the same owner entity*.
- The partial key attribute is underlined with a dashed or dotted line.

A Sample Database Application: COMPANY DATABASE

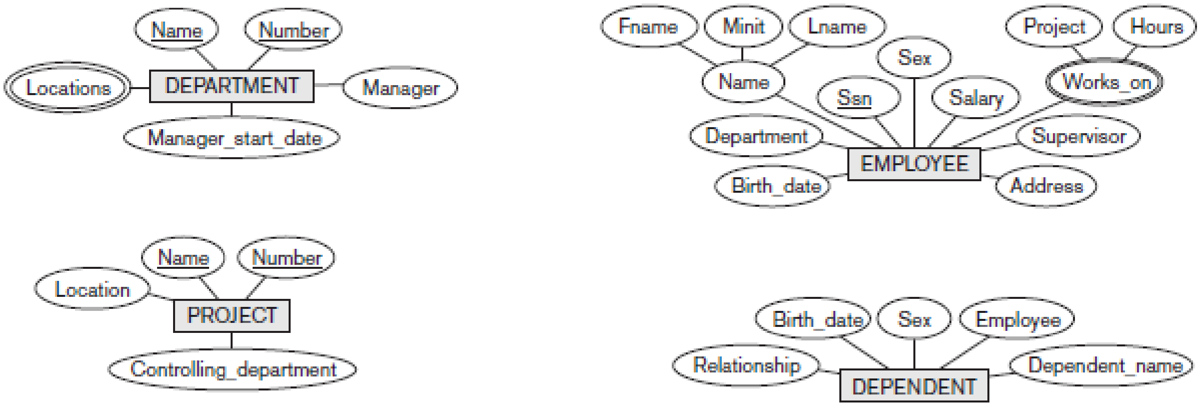
The COMPANY database keeps track of a company's employees, departments, and

projects.

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- The database will store each employee's name, Social Security number,2 address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. It is required to keep track of the current number of hours per week that an employee works on each project, as well as the direct supervisor of each employee (who is another employee).
- The database will keep track of the dependents of each employee for insurance purposes, including each dependent's first name, sex, birth date, and relationship to the employee.

Initial 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, and DEPENDENT. Their initial attributes are derived from the requirements description. Initial design is shown in the figure



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

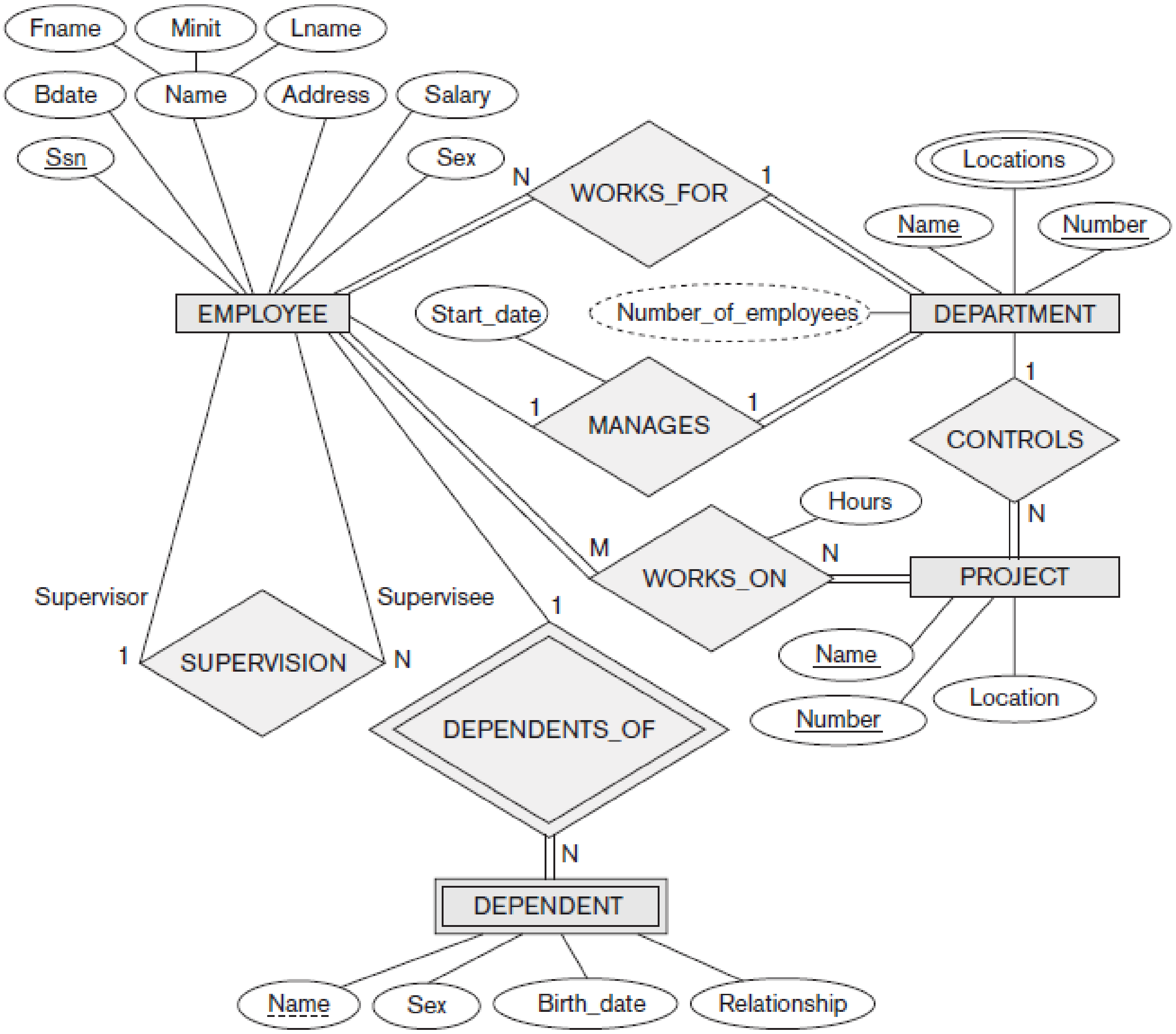
Refining the COMPANY database schema by introducing relationships

By refining 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)

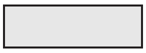
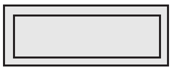
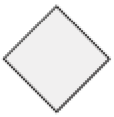
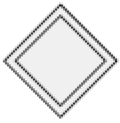
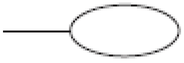
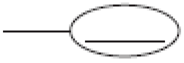
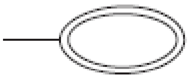
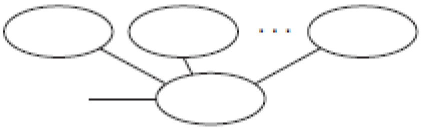
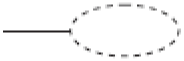
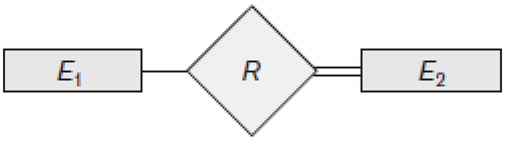
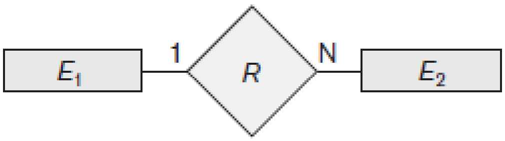
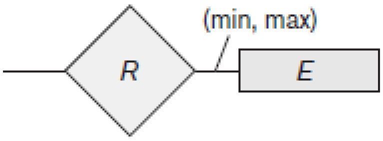
In the refined design, some attributes from the initial entity types are refined into relationships:

Manager of DEPARTMENT -> MANAGES
Works_on of EMPLOYEE -> WORKS_ON
Department of EMPLOYEE -> WORKS_FOR



An ER schema diagram for the COMPANY database.

Summary of the notation for ER diagrams

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Indentifying 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

Question Bank

1. Define the following terms:
   data, *database*, *DBMS*, *database system*, *database catalog*, *program-data independence*, *user view*, *DBA*, *end user*, *canned transaction* and *meta-data*.
2. Discuss the main characteristics of the database approach and how it differs from traditional file systems.
3. What are the responsibilities of the DBA and the database designers?
4. What are the different types of database end users? Discuss the main activities of each.
5. Discuss the capabilities that should be provided by a DBMS.
6. Discuss the differences between database systems and File system.
7. Discuss the advantages and disadvantages of DBMS.
8. Discuss applications of database system.
9. Define the following terms:
   Data model, database schema, database state, internal schema, conceptual schema, external schema, data independence, DDL, DML, SDL, VDL, query language, host language, data sublanguage.
10. Discuss the main categories of data models.
11. What is the difference between a database schema and a database state?
12. Describe the three-schema architecture. Why do we need mappings among schema levels?
13. What is the difference between logical data independence and physical data independence? Which one is harder to achieve? Why?
14. Explain database Languages.
15. Discuss the different types of user-friendly interfaces and the types of users who typically use each.
16. Explain the important component modules of a DBMS and their interactions with diagram.
17. Discuss database system architectures. What is the difference between the two-tier and three-tier client/server architectures?
18. Discuss the use of high-level conceptual data model in the database design process.
19. List the various cases where use of a NULL value would be appropriate.
20. Define the following terms:
    Entity, attribute, attribute value, relationship instance, composite attribute, multivalued attribute, derived attribute, complex attribute, key attribute, and domain
21. What is an entity type? What is an entity set? Explain the differences among an entity, an entity type, and an entity set.
22. What is a relationship type? Explain the differences among a relationship type, and a relationship set.
23. What is a role name? When is it necessary to use role names in the description of relationship types?
24. Describe the two alternatives for specifying structural constraints on relationship types. What are the advantages and disadvantages of each?
25. Under what conditions can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity types?
26. What is meant by a recursive relationship type? Give some examples of recursive relationship types.
27. When is the concept of a weak entity used in data modeling? Define the terms owner entity type, weak entity type, identifying relationship type, and partial key.
28. Can an identifying relationship of a weak entity type be of a degree greater than two? Give examples to illustrate your answer.
29. Discuss the naming conventions used for ER schema diagrams.

Module 2

Relational Model

- Relational Model Concepts
- Relational Model Constraints and relational database schemas
- Update operations
- Transactions
- Dealing with constraint violations.

Relational Algebra

- Unary and Binary relational operations
- Additional relational operations (aggregate, grouping, etc.)
- Examples of Queries in relational algebra.

Mapping Conceptual Design into a Logical Design

- Relational Database Design using ER-to-Relational mapping.

Textbook 1: Ch 5.1 to 5.3,8.1 to 8.5, 6.1 to 6.5; Textbook 2: 3.5

The Relational Data Model and Relational Database Constraints

Introduction

- The **relational data model** was first introduced by **Ted Codd of IBM Research in 1970**, and it was popular due to its simplicity and mathematical foundation.
- The model uses the concept of a *mathematical relation*—tables of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic.
- The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS.
- Since then, the model has been implemented in a large number of commercial systems, as well as a number of open source systems.
- Current popular commercial relational DBMSs (RDBMSs) include **DB2 (from IBM), Oracle (from Oracle), Sybase DBMS (now from SAP), and SQLServer and Microsoft Access (from Microsoft).**
- In addition, several open source systems, such as **MySQL, PostgreSQL**, **SQLite and MariaDB** are available.

Relational Model Concepts

- The relational model represents the **database as a collection of *relations***.
- Each relation resembles a table of values or, to some extent, a *flat* file of records.
- A relation is thought of as a **table** of values, each row in the table represents a collection of related data values.
- A row represents a fact that typically corresponds to a real-world entity or relationship.
- The table name and column names are used to help to interpret the meaning of the values in each row.
- For example, STUDENT Relation each row represents facts about a particular student entity. The column names—Name, Student_number, Class, and Major—specify how to interpret the data values in each row, based on the column each value is in.
- All values in a column are of the same data type.
- In the relational model terminology, a row is called a ***tuple***, a column header is called an ***attribute***, and the table is called a ***relation***. The data type describing the types of values that can appear in each column is represented by a ***domain*** of possible values.

Domains

A **domain *D*** is a set of atomic values, means that each value in the domain is indivisible. A domain is defined by specifying a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values.

A domain is thus given a **name, data type, and format**. Additional information can also be given, a numeric domain such as Person_weights should have the units of measurement, such as pounds or kilograms.

- Phone_numbers. The set of ten-digit phone numbers valid in the India.
- Social_security_numbers. The set of valid nine-digit Social Security numbers.
- Names: The set of character strings that represent names of persons.
- Employee_ages. Possible ages of employees in a company; each must be an integer value between 15 and 80.
- A data type or format is also specified for each domain.
- Example, the data type for the domain Phone_numbers can be declared as a character string of the form (ddd)ddd-dddd, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code.

Relation Schema

- A **relation schema** *R*, denoted by *R*(*A*1, *A*2, … , *An*), is made up of a relation name *R* and a list of attributes, *A*1, *A*2, … , *An*. A relation schema is used to *describe* a relation; *R* is called the **name** of this relation.
- Each **attribute** *Ai* is the name of a role played by some domain *D* in the relation schema *R*.
- *D* is called the **domain** of *Ai* and is denoted by **dom**(*Ai*).
- The **degree** (or **arity**) of a relation is the number of attributes *n* of its relation schema.
- A STUDENT relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

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

or

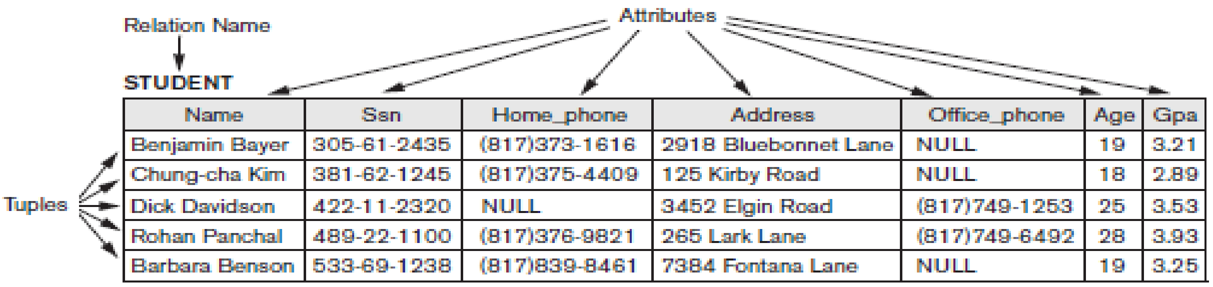
STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

More precisely, we can specify defined domains for some of the attributes of the STUDENT relation:

dom(Name) = Names; dom(Ssn) = Social_security_numbers; dom(Office_phone) = Phone_numbers

Relation State

- A **relation** (or **relation state**) *r* of the relation schema *R*(*A*1, *A*2, … , *An*), also denoted by *r*(*R*), is a set of *n*-tuples *r* = {*t*1, *t*2, … , *tm*}. Each ***n*-tuple** *t* is an ordered list of *n* values *t* =<*v*1, *v*2, ... , *vn*>, where each value *vi*, 1 ≤ *i* ≤ *n*, is an element of dom (*Ai*) or is a special NULL value.
- The *i*th value in tuple *t*, which corresponds to the attribute *Ai*, is referred to as *t*[*Ai*] or *t*.*Ai* or *t*[*i*].
- The relation schema *R* is **relation intension** and relation state *r*(*R*) is**relation extension.**



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

Characteristics of Relations

- A relation is defined as a *set* of tuples. Hence, tuples in a relation do not have any particular order. A relation is not sensitive to the ordering of tuples.
- The ordering of attributes is *not* important, because the *attribute name* appears with its *value*.
- Each value in a tuple is an **atomic** value;
- Composite and multivalued attributes are not allowed.
- A special value, called NULL, is used to represent the values of attributes that may be unknown or may not apply to a tuple.

NOTE

- A relation schema *R* of degree *n* is denoted by $R(A_1, A_2, \dots, A_n)$.
- The uppercase letters *Q*, *R*, *S* denote relation names.
- The lowercase letters *q*, *r*, *s* denote relation states.
- The letters *t*, *u*, *v* denote tuples.

Relational Databases and Relational Database Schemas

A relational database usually contains many relations, with tuples in relations that are related in various ways.
A **relational database schema S** is a set of relation schemas S = {R1, R2, … , Rm} and a set of integrity constraints IC.
A relational database state DB of S is a set of relation states DB = {r1, r2, … , rm} such that each ri is a state of Ri and such that the ri relation states satisfy the integrity constraints specified in IC.

Example: Relational database schema
COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}

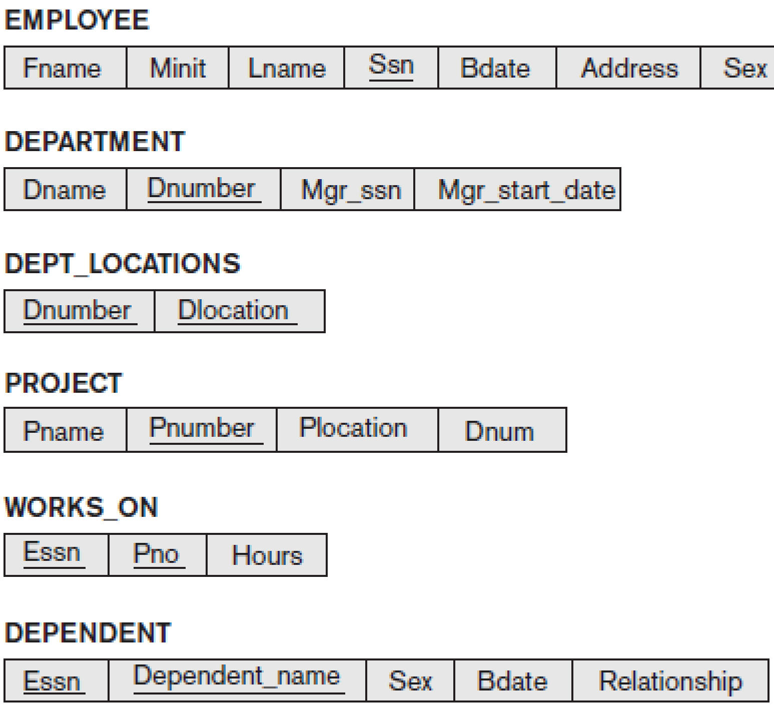
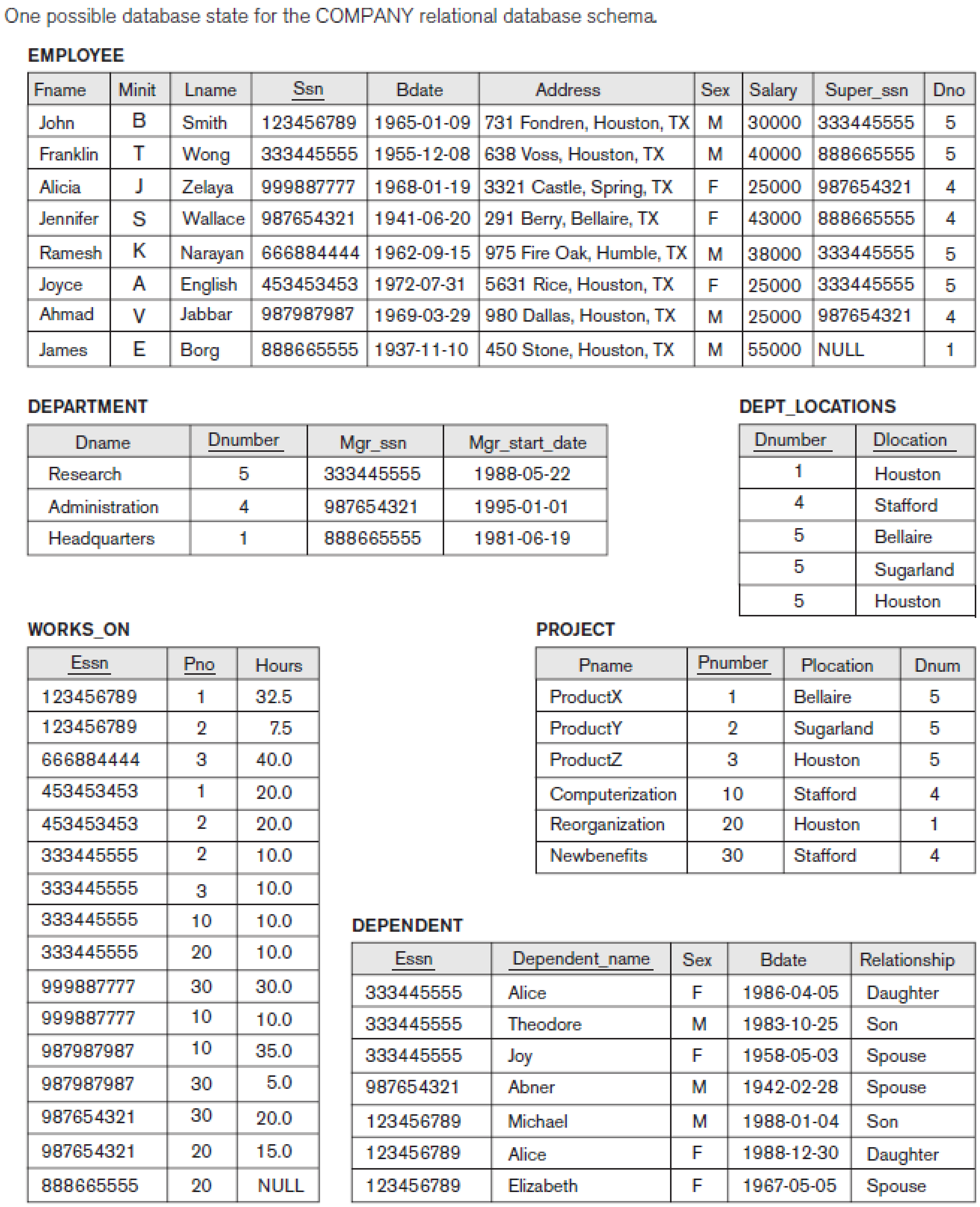


Figure 5.5
Schema diagram for the COMPANY relational database schema.

Note: Include the arrows for foreign key representation.

NOTE: Relational databases implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called **not valid**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**



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

Relational Model Constraints

Various restrictions on data can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

1. Inherent model-based constraints or **implicit constraints**: Constraints those are inherent in the data model. Example:- The characteristics of relations, like a relation cannot have duplicate tuples, ordering of tuples is not important, composite and multivalued attributes are not allowed.

2. Schema-based constraints or explicit constraints: Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL. Example: domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

3. Application-based or semantic constraints or business rules: Constraints that cannot be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. Example: Assertions and triggers in SQL.

Types of schema-based constraints or explicit constraints

1. Domain Constraints
2. Key Constraints
3. Constraints on NULL Values
4. Entity Integrity
5. Referential Integrity and Foreign Keys

1. Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain dom(A).
The data types associated with domains include integers (short integer, integer, and long integer) and real numbers (float,double-precision float).Characters, Booleans, fixed and variable-length strings, and date, time, timestamp.

2. Key Constraints

Superkey:
There are **subsets of attributes** of a relation schema *R* with the property that no two tuples in any relation state *r* of *R* should have the same combination of values for these attributes. Such subset of attributes are denoted by SK; then for any two *distinct* tuples *t*1 and *t*2 in a relation state *r* of *R*, then have the constraint that:

$$t1[SK]= t2[SK]$$

Any such set of attributes SK is called a **superkey** of the relation schema *R*.
A **superkey** SK specifies a *uniqueness constraint* that no two distinct tuples in any state *r* of *R* can have the same value for SK. Every relation has at least one default superkey— the set of all its attributes. A superkey can have redundant attributes.

Key:

A **key** *k* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K′* that is not a superkey of *R* any more.

Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This *uniqueness* property also applies to a superkey.
2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold.

Example: In STUDENT relation, attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.
Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey.
However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey.

NOTE

- A key is a superkey but not vice versa.
- A superkey may be a key (if it is minimal) or may not be a key (if it is not minimal).
- In general, any superkey formed from a single attribute is also a key.
- A key with multiple attributes must require all its attributes together to have the uniqueness property.
- The value of a key attribute can be used to identify uniquely each tuple in the relation.
- A key is determined from the meaning of the attributes, and the property is time-invariant: It must continue to hold when we insert new tuples in the relation.

Candidate key

A relation schema may have more than one key. In this case, each of the keys is called a candidate key.
Example, the CAR relation has two candidate keys: License_number and Engine_serial_number.

Primary key

Key whose values are used to identify tuples in the relation is called primary key. It is common to designate one of the candidate keys as the primary key of the relation.

NOTE

- A relation schema has several candidate keys, the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes.
- The other candidate keys are designated as **unique keys** and are not underlined.

3. Constraints on NULL Values

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

4. Entity Integrity

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples.
Example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations.

5. Referential Integrity and Foreign Keys

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations.

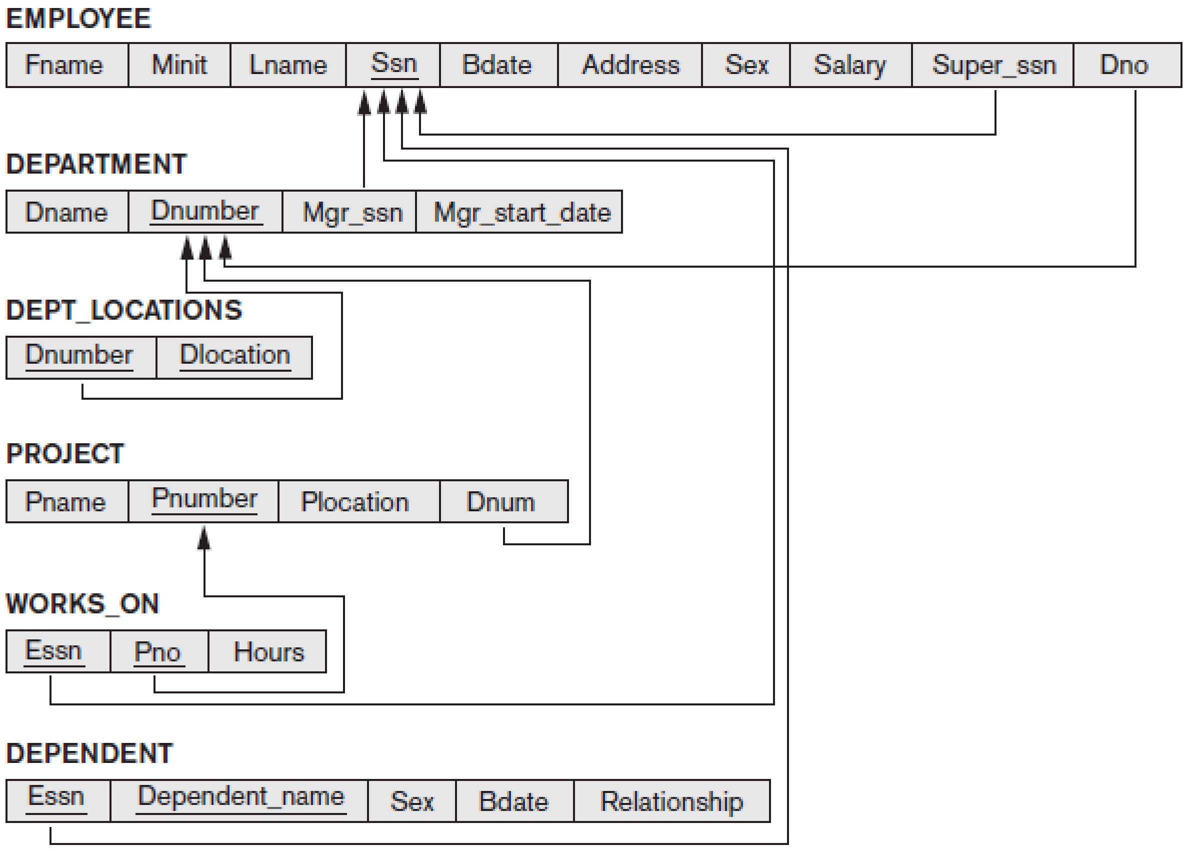
The **referential integrity constraint** states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

Example: Attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

Foreign key: The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R1 and R2. A set of attributes FK in relation schema R1 is a foreign key of R1 that references relation R2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.
2. A value of FK in a tuple t1 of the current state r1(R1) either occurs as a value of PK for some tuple t2 in the current state r2(R2) or is NULL.

- t1[FK] = t2[PK], and that is the tuple t1 references or refers to the tuple t2.
- R1 is called the referencing relation and R2 is the referenced relation.
- If these two conditions hold, a referential integrity constraint from R1 to R2 is said to hold.
- In a database of many relations, there are usually many referential integrity constraints.
- Referential integrity constraints typically arise from the relationships among the entities represented by the relation schemas.
- **Example**, consider the COMPANY database, EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation.
- Means a value of Dno in any tuple t1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t2 of the DEPARTMENT relation, or the value of Dno can be NULL if the employee does not belong to a department or will be assigned to a department later.
- Foreign key can *refer to its own relation.* For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself.
- *Diagrammatically referential integrity constraints* is *displayed* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation.



Referential integrity constraints displayed on the COMPANY relational database schema.

Other Types of Constraints

Semantic integrity constraints are not part of the DDL and have to be specified and enforced in a different way. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee's supervisor* and *the maximum number of hours an employee can work on all projects per week is 56*. Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**.

Mechanisms called **triggers** and **assertions** can be used in SQL, through the CREATE ASSERTION and CREATE TRIGGER statements, to specify some of these constraints.

The types of constraints discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy.

Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.1. An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically enforced by the application programs or specified using active rules and triggers.

Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* There are three basic **update** operations that can change the states of relations in the database:

Insert: insert new data**,**

Delete: delete old data**, and**

Update (or Modify): modify existing data records

Insert is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database
schema should not be violated.

The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple *t* that is to be inserted into a relation *R*. Insert can violate any of the four types of constraints.

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

Examples to illustrate violation of insertion operation:

INSERT INTO EMPLOYEE VALUES ('&Fname', '&Minit', '&Lname', 'SSN', 'Bdate', 'Address', 'Sex', '&Salary','SuperSsn', '&Dno');

- ***Operation*:** Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE. *Result*: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

- *Operation*: Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.
  Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

- *Operation*: Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
  Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

- *Operation*: Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
  Result: This insertion satisfies all constraints, so it is acceptable.

NOTE: If an insertion violates one or more constraints, the default option is to *reject the insertion.* Another option is to attempt to *correct the reason for rejecting the insertion*.

The Delete Operation

The **Delete** operation deletes the existing records and can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. Examples:

- *Operation*: Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.
  Result: This deletion is acceptable and deletes exactly one tuple.
- *Operation*: Delete the EMPLOYEE tuple with Ssn = '999887777'.
  Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

Several options are available if a deletion operation causes a violation.

- **Restrict,** is to *reject the deletion.*
- **Cascade,** is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = '999887777'.
- **set null** or **set default,** is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple.

NOTE:

- Notice that if a referencing attribute that causes a violation is *part of the primary key*, it *cannot* be set to NULL; otherwise, it would violate entity integrity.
- Combinations of these three options are also possible.
- In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to *specify which of the options* applies in case of a violation of the constraint by Delete and a violation caused by Update.

The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Update can violate any of the four types of constraints like insert. Examples.

- *Operation*: Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
  Result: Acceptable.
- *Operation*: Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.
  Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Similar options exist to deal with referential integrity violations caused by Update: **Restrict, Cascade and set null** or **set default.**

NOTE:

- Updating an attribute that is neither part of a primary key nor part of a foreign key usually causes no problems;
- the DBMS need only check to confirm that the new value is of the correct data type and domain.
- Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples.
- If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL).

Exercise Problem

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)

1. **Draw a ER diagram for this database.**
2. **Draw a relational schema diagram specifying the foreign keys for this schema**

Answer

The Attribute SSN of relation ENROLL that references relation STUDENT

The Attribute Course# of relation ENROLL that references relation COURSE

The Attribute Course# of relation BOOK_ADOPTION that references relation COURSE

The Attribute Book_ISBN of relation BOOK_ADOPTION that references relation TEXT

Designing ER diagram involves the following:

1. Identifying strong entities and their attributes.
2. Choosing appropriate names for the relations and their attributes.
3. Identify implicit relationships among the various entity types and represent relationship type
4. Identifying the candidate keys and choosing a primary key for each relation.

Designing the relational database schema involves the following:

1. Deciding which attributes belong together in each relation
2. Choosing appropriate names for the relations and their attributes.

3. Specifying the domains and data types of various attributes.
4. Identifying the candidate keys and choosing a primary key for each relation.
5. Specifying all foreign keys.

Relational Algebra

The two *formal languages* for the relational model: the relational algebra and the relational calculus. Historically, the relational algebra and calculus were developed before the SQL language. SQL is primarily based on concepts from relational calculus and has been extended to incorporate some concepts from relational algebra as well. Most relational DBMSs use SQL as their language.

- The basic set of operations for the formal relational model is the **relational algebra**.
- These operations enable a user to specify basic retrieval requests as relational algebra expressions.
- A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query.
- It provides a formal foundation for relational model operations.
- It is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs)
- Some of its concepts are incorporated into the SQL standard query language for RDBMSs.
- The **relational calculus** provides a higher-level declarative language for specifying relational queries.
- In a relational calculus expression, there is no order of operations to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus.
- The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in a variation of relational calculus known as the tuple relational calculus.
- The relational algebra operations can be divided into two groups: **set operations (**UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN) and other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others.

Unary Relational Operations: SELECT and PROJECT

The SELECT Operation

- The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**.
- The SELECT operation acts as a *filter* that keeps only those tuples that satisfy a qualifying condition.
- The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are filtered out.
- The SELECT operator is **unary**; that is, it is applied to a single relation.
- The relation resulting from the SELECT operation has the *same attributes* as *R* that is **degree** of the relation resulting is the same as the degree of *R*.

- In general, the SELECT operation is denoted by

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

  - the symbol σ (sigma) is used to denote the SELECT operator
  - selection condition is a Boolean expression (condition) specified on the attributes of relation *R*, which can be of the form

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

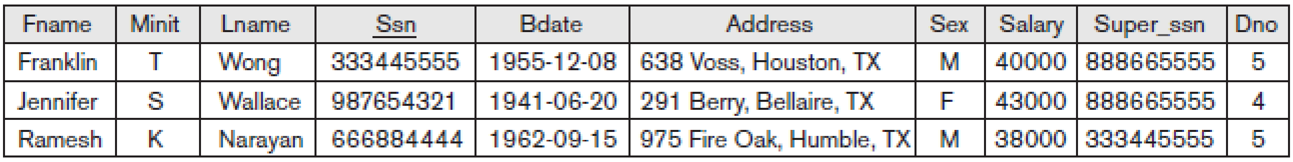
    Or <attribute name> <comparison op> <attribute name>
  - where <attribute name> is the name of an attribute of *R*, <comparison op> is normally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant value from the attribute domain.
  - Clauses can be connected by the standard Boolean operators *and*, *or*, and *not*

Example: 1. To select the EMPLOYEE tuples whose department is 4.

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

Example 2: To select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

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



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

NOTE:

- The selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple.
- The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in *R*. That is, $|\sigma c\ (R)| \leq |R|$ for any condition *C*.
- The SELECT operation is **commutative**; we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition;

  σ<cond1>(σ<cond2>(... (σ<cond*n*>(*R*)) ...)) = σ<cond1> **AND**<cond2> **AND**...**AND** <cond*n*>*(R)*

The PROJECT Operation

- The PROJECT operation selects certain *columns* from the table and discards the other columns.
- If only certain attributes of a relation are needed then the PROJECT operation is used to *project* the relation over these attributes only.
- It can be visualized as a *vertical partition* of the relation into two relations: one has the needed columns and contains the result of the operation, and the other contains the discarded columns.
- The resulting relation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.
- The general form of the PROJECT operation is

$$\pi_{\text{<attribute list>}}(R)$$

- π (pi) is the symbol used to represent the PROJECT operation
- <attribute list> is the desired sublist of attributes from the attributes of relation *R*

Example: To list each employee's first and last name and salary:

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

NOTE:

- The PROJECT operation removes any duplicate tuples, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as duplicate elimination.
- The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in *R*.
- Relational algebra expression can be written as an **in-line expression**, as follows:

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

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

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$$

The RENAME Operation

RENAME operation can rename either the relation name or the attribute names, or both—as a unary operator. The general is denoted by any of the following three forms:

$$\rho_{S(B1, B2, \ldots, Bn)}(R) \quad \text{or} \quad \rho_{S}(R) \quad \text{or} \quad \rho_{(B1, B2, \ldots, Bn)}(R)$$

- where the symbol ρ (rho) is used to denote the RENAME operator
- *S* is the new relation name
- *B*1, *B*2, … , *Bn* are the new attribute names.
- The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of *R* are (*A*1, *A*2, … , *An*) in that order, then each *Ai* is renamed as *Bi*.

Example:

$$\text{TEMP} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$R(\text{First\_name, Last\_name, Salary}) \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{TEMP})$$

Relational Algebra Operations from Set Theory

- Relational algebra operations based on the standard mathematical operations on sets: UNION, INTERSECTION, and SET DIFFERENCE (also called MINUS or EXCEPT) are binary operations; that is, each is applied to two sets (of tuples).
- The two relations on which any of these three operations are applied must have the same type of tuples; this condition has been called **union compatibility or type compatibility.**

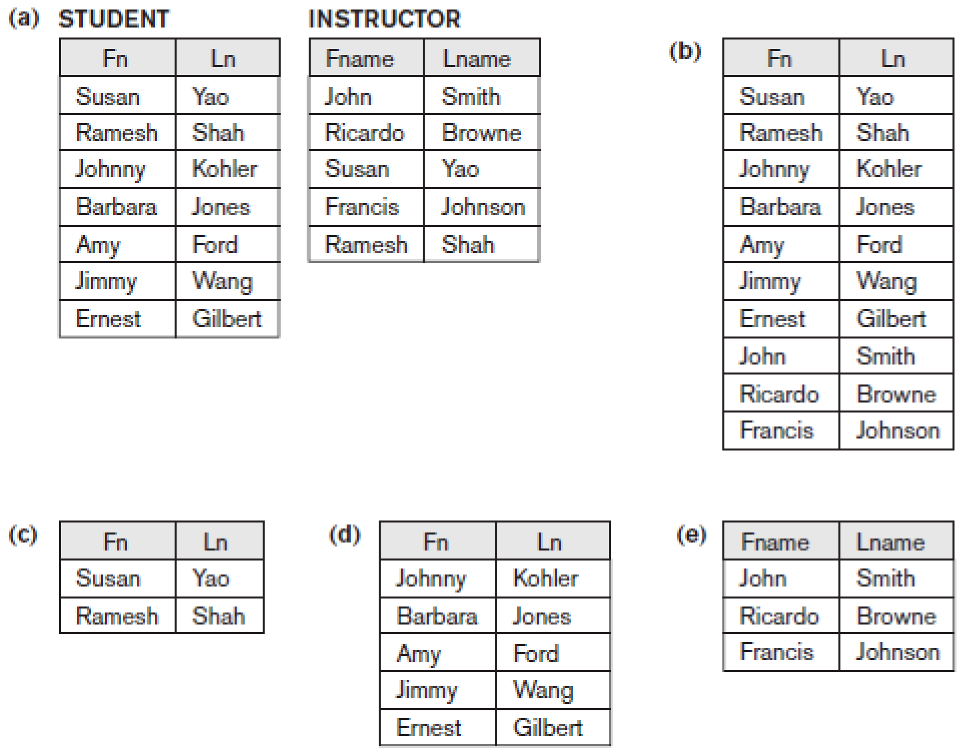
- Two relations R(A1, A2, … , An) and S(B1, B2, … , Bn) are said to be union compatible (or type compatible) if they have the same degree n and if dom(Ai) = dom(Bi) for 1 ≤ i ≤ n. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

The three operations UNION, INTERSECTION, and SET DIFFERENCE are defined on two union-compatible relations R and S as follows:

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

INTERSECTION: The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S.
SET DIFFERENCE (or MINUS): The result of this operation, denoted by R – S, is a relation that includes all tuples that are in R but not in S.

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



(a) STUDENT

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

INSTRUCTOR

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

(b)

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

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

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

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

NOTE:

- Both UNION and INTERSECTION are ***commutative operations***; $R \cup S = S \cup R$ and $R \cap S = S \cap R$
- Both UNION and INTERSECTION can be treated as *n*-ary operations applicable to any number of relations because both are also ***associative operations****;* $R \cup (S \cup T) = (R \cup S) \cup T$ and $(R \cap S) \cap T = R \cap (S \cap T)$
- The MINUS operation is ***not commutative****;* that is, in general, $R - S \neq S - R$
- Note that INTERSECTION can be expressed in terms of union and set difference as follows:

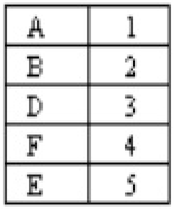
$$R \cap S = ((R \cup S) - (R - S)) - (S - R)$$

- In SQL, these are implemented using—UNION, INTERSECT, and EXCEPT operations. In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

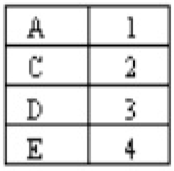
- **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by ×. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible.
- In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A1, A2, \dots, An) \times S(B1, B2, \dots, Bm)$ is a relation Q with degree $n + m$ attributes $Q(A1, A2, \dots, An, B1, B2, \dots, Bm)$, in that order.The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has nR tuples (denoted as $|R| = nR$), and S has nS tuples, then $R \times S$ will have $nR * nS$ tuples.

R



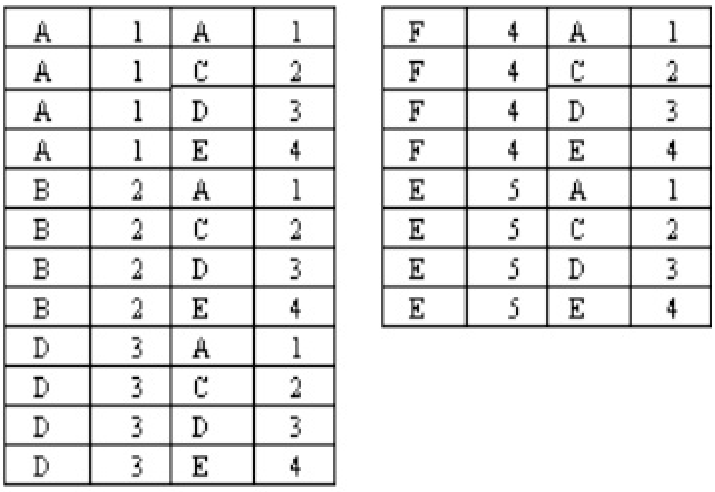
A	1
B	2
D	3
F	4
E	5

S



A	1
C	2
D	3
E	4

R CROSS S



A	1	A	1
A	1	C	2
A	1	D	3
A	1	E	4
B	2	A	1
B	2	C	2
B	2	D	3
B	2	E	4
D	3	A	1
D	3	C	2
D	3	D	3
D	3	E	4
F	4	A	1
F	4	C	2
F	4	D	3
F	4	E	4
E	5	A	1
E	5	C	2
E	5	D	3
E	5	E	4

Binary Relational Operations: JOIN and DIVISION

The JOIN Operation

The JOIN operation is very important for any relational database with more than a one relation because it allows us to process relationships among relations.

The **JOIN operation**, denoted by ⋈ , on two relations R(A1, A2, … , An) and S(B1, B2, … , Bm) combines related tuples from two relations into single "longer" tuples that satisfy the join condition. The general form

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

- The result of the JOIN is a relation Q with $n + m$ attributes $Q(A1, A2, \dots, An, B1, B2, \dots, Bm)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition*.

- A general join condition: <condition> **AND** <condition> **AND … AND** <condition> where each <condition> is of the form *Ai* θ *Bj*, *Ai* is an attribute of *R*, *Bj* is an attribute of *S*, *Ai* and *Bj* have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}.
- A JOIN operation with such a general join condition is called a **THETA JOIN**.
- Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result.

Example: Suppose that we want to retrieve the name of the manager of each department.

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

To get the manager's name, combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple.

The EQUIJOIN

EQUIJOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an EQUIJOIN.

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

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

Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple. For example, the values of the attributes Dnumber and Dno are identical in every tuple of RESEARCH_EMP because the equality join condition specified on these two attributes requires the values to be identical in every tuple in the result.

NATURAL JOIN

NATURAL JOIN—denoted by *, requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first. NATURAL JOIN was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

Example 1, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS,

DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS

Example 2: Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.First rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

PROJ_DEPT ← PROJECT * ρ(Dname, Dnum, Mgr_ssn, Mgr_start_date)(DEPARTMENT)

The attribute Dnum is called the join attribute for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is only one join attribute value is kept.

ER-to-Relational Mapping

Step 1

» For each regular entity type E in the ER schema, create a relation R that includes all the simple attributes of E
» For composite attributes, use the simple component attributes.
» Choose one of the key attributes of E as the primary key for R.
» If the chosen key was a composite, the set of simple attributes that form it will together be the primary key of R.

Step 2

» For each weak entity type W in the ER schema with owner entity type E, create a relation R that includes all simple attributes (or simple components of composites) of W.
» Include as foreign key attributes of R the primary key attribute of the relation that corresponds to the owner entity type E.
» The primary key of R is the combination of the primary key of the owner and the partial key of the weak entity type, if any.

Step 3

» 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.
» Choose one of the relations (say S). Include as a foreign key in S the primary key of T.
» It is better to choose an entity type with total participation in R for the role of S.
» Include the simple attributes of R as attributes of S.

Step 4

» For each binary 1:N relationship type R, identify the relation S that represents the entity type participating at the N-side of the relationship
» Include as a foreign key in S the primary key of the relation T that represents the other entity type participating in R.
» Include the simple attributes of R as attributes of S.

Step 5

» For each binary M:N relationship type R, create a new relation S to represent R.
» Include as foreign key attributes in S the primary keys of the participating entity types.
» Their combination will form the primary key.
» Include any attributes of R as attributes of S.

Step 6

» For each multi-valued attribute A, create a new relation R.

» R will include an attribute corresponding to A, plus the primary key attribute K of the relation that has A as an attribute.
» The primary key of R is the combination of A and K.

Step 7

» For each n-ary relationship type R, where n>2, create a new relation S.
» Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types
» Include any attributes of R.
» The primary key of S is usually a combination of all the foreign keys in S.

Question Bank

1. Define the following terms as they apply to the relational model of data:
   domain, attribute, relation schema, relation state, degree of a relation, relational database schema, and relational database state.
2. Define the following: superkey, key, candidate key, alternate key, primary key
3. Discuss the important characteristics of Relational Data Model.(jan-2018)
4. Explain different Relational Model Constraints with examples.
5. Explain the following terms with example.i) Cardinality ratio ii) Total participation iii) Partial participation.
   or
   Explain the structural constraints.
6. Discuss the update operations (insert, delete and update) on Relational Model and explain the constraints that may be violated.
   Or
   What are the basic operations that can change the state of relations in the database? Explain how the basic operations deal with constrain violations.(Jan-2018)
7. Define *foreign key*. What is this concept used for?
8. Explain constraints are key, entity integrity, and referential integrity constraints.
9. Explain the operations of relational algebra(select, project and rename).
10. Explain the relational model operation from set theory(union,intersection,minus).
11. Explain Binary Relational Operation: DIVISION
12. Explain JOIN and its types with examples. Differentiate between equi join and natural join
13. What are outer join operations and how are they different from inner join operations?
14. How is Aggregate Functions and Grouping done in relation algebra?
15. What is cartesian product ? How is it different from join operation?
16. Describe the steps of ER-To-Relational mapping algorithm.(jan-2018)

Module 3

SQL

- SQL data definition and data types
- Specifying constraints in SQL
- Retrieval queries in SQL
- INSERT, DELETE, and UPDATE statements in SQL
- Additional features of SQL.

Advances Queries

- More complex SQL retrieval queries
- Specifying constraints as assertions and action triggers
- Views in SQL
- Schema change statements in SQL

Database Application Development

- Accessing databases from applications
- An introduction to JDBC
- JDBC classes and interfaces

- Stored procedures

Internet Applications

- The three-Tier application architecture

Textbook 1: Ch7.1 to 7.4; Textbook 2: 6.1 to 6.6, 7.5 to 7.7.

Basic SQL

- Structured Query Language (SQL) is a standard language for relational databases.

- Commercial RDBMSs that support SQL makes it easy to convert from one RDBMS to another because they all follow the same language standard.

- SQL is nonprocedural language meaning the user specifies what needs to be done leaving the actual optimization and decisions on how to execute the query to the DBMS.

- Relational Algebra provides a formal foundation for SQL which has a more user friendly syntax than Relational Algebra.

- SQL is a comprehensive database language. It supports:

  - Data Definition Language (DDL)
  - Data Manipulation Language (DML)
  - Views
  - Security and Authorization
  - Integrity constraints
  - Transaction control
  - Rules for embedding SQL statements into a general purpose programming language such as Java , COBOL or C/C++

- The standard version we will learn in this chapter is SQL 99 (also called SQL3)
- Previous versions are SQL 86 (or SQL1), SQL 92 (or SQL2)

- ***SQL Data Definitions and Data Types:***

- SQL uses the terms table, row and column for the formal model terms relation, tuple and attribute.

- The main SQL command for data definition is the **CREATE** statement which can be used to create schemas, tables, domains, views, assertions and triggers.

- Schema and Catalog Concepts in SQL

- The concept of a relational database schema started in SQL 92.

- An SQL schema is used to group together tables and other constructs that belong to the same data base application.

- An SQL schema is identified by a schema name and authorization number to indicate the user or account who owns the schema.

- A schema is created using the **CREATE SCHEMA** command as in:
  CREATE SCHEMA COMPANY AUTHORIZATION JSMITH

- CREATE TABLE Command

- The **CREATE TABLE** command is used to create a new table by giving it a name and specifying its attributes and initial constraints.
- For example, to create a table called **EMPLOYEE** and attach it to the **COMPANY** schema created above:

```
CREATE TABLE COMPANY.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(1),
   SALARY         DECIMAL(10,2),
    SUPERSSN      CHAR(9),
    DNO           INT                  NOT NULL,
    PRIMARY KEY (SSN),
    FOREIGN KEY (SUPERSSN) REFERENCES
    EMPLOYEE(SSN),
    FOREIGN KEY (DNO) REFERENCES
    DEPARTMENT(DNUMBER)
  );
```

- Tables created using the **CREATE TABLE** command are called base tables because they are physically stored as files by the DBMS.

- Data Types and Domains

- The basic data types are numeric, character, strings, bit strings, boolean, date and time.

- Numeric data types

  - INTEGER or INT and SMALLINT for integer numbers.

  - FLOAT or REAL and DOUBLE PRECISION for floating-point (real) numbers.

  - Formatted numbers can be declared using DECIMAL(i,j), DEC(i,j) or NUMERIC(i,j) where i is the total number of decimal digits and j is the number of digits after the decimal point.

- Character-string data types

  - Fixed length: CHAR(n) or CHARACTER(n) where n is the number of characters.

  - Variable length: VARCHAR(n) or CHARVARAYING(n) or CHARACTER VARAYING(n) where n is the maximum of characters.

  - String values must be enclosed between single quotes (apostrophes).

  - String values are case sensitive meaning that *' SMITH ‘, 'Smith'* and *'smith'* are all different.

  - For fixed-length strings, unused characters are padded with blank character which are ignored when strings are compared.

  - Strings are alphabetically ordered therefore STR1 < STR2 if STR1 appears before STR2 in alphabetic order.

  - The ‖ is used for string concatenation therefore *'abc'* ‖ *' xyz'* is *'abcxyz'*.

- Bit-string data types

  - Fixed length: BIT(n)
  - Varying length: BITVARYING(n)
  - Bit strings are enclosed between single quotes but are preceded by B to distinguish them from character strings
    Example: B '10101'

- Boolean for TRUE / FALSE values

- Date and Time :

  - Date is represented as YYYY-MM-DD.
  - Time is represented as HH:MM:SS.
  - Date values are preceded by Keyword DATE.
    Example: DATE ' 2002-09-27’
  - Time values are preceded by Keyword TIME.
    Example: TIME '09:12:47'

  - Timestamp to include both DATE and TIME fields plus a minimum of six

positions for decimal fractions of seconds.
- Timestamp values must be preceded by Keyword TIMESTAMP

Example: TIMESTAMP' 2002:09:27 09:12:47 648 302'

- Constraints in SQL

- Attribute Constraints and Attribute Defaults

- By default, SQL allows NULL values for attributes.
- To Disallow NULL values, the attribute should be declare using the **NOT NULL** constraint as follows:

```
DNAME VARCHAR(15) NOT NULL
```

- Default values can also be specified using the DEFAULT Keyword as follows:

```
DNO     INT     NOT NULL    DEFAULT  1
```

- If no default value specified a default value of NULL will used for attributes that do not have the **NOT NULL** constraint.

- A **CHECK** constraint is used to restrict attribute values:

```
DNUMBER INT  NOT NULL CHECK(DNUMBER>0 AND DNUMBER<21)
```

- A **CHECK** constraint can also be used to restrict domain values:

```
CREATE   DOMAIN   DNUM   AS  INTEGER CHECK(DNUM>0 AND DNUM<21)
```

- Key and Referential Integrity Constraints

- **PRIMARY KEY** clause is used to specify one or more attributes that make up the primary key as follows:

```
DNBUMBER    INT     PRIMARY KEY;
```

- Alternatively, as shown in Figure 8.1, it can be done as follows:

```
DNBUMBER    INT     NUT NULL,
 .
 .
 .
PRIMARY KEY(DNUMBER);
```

- Composite primary keys can be declared on follows

```
DNUMBER INT NOT NULL,
DLOCATION VARCHAR(5) NOT NULL,
PRIMARY KEY (DNUMBER, DLOCATION)
```

- The **FOREIGN KEY** clause is used to specify foreign keys as shown in the EMPLOYEE table:

```
SUPERSSN CHAR(9),
DNO INT NOT NULL,
FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE (SSN),
FOREIGN KEY (DNO) REFERENCES DEPARTMENT (DNUMBER);
```

- The **FOREIGN KEY** clause can contain the following options:

- **ON DELETE SET NULL**: if the corresponding primary key is deleted, set the foreign key to NULL. As shown in Figure 8.2 this option is specified as follows.

FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE (SSN) ON DELETE SET NULL

- For example as shown in Company database, note that employee John Smith is supervised by Franklin Wong. If the tuple for Franklin Wong is deleted, set the SUPERSSN for John Smith to NULL.

- **ON DELETE CASCADE**: if the tuple for the corresponding primary key is deleted, delete all tuples whose foreign keys reference this primary key. As shown in Figure 8.2 this option can be specified as follows:

FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE (SSN) ON DELETE CASCADE

- For example, in Figure 5.6, if the tuple for the supervisor Franklin Wong is deleted, the tuples for John Smith, Ramesh & Joyce English will also be deleted.

- **ON DELETE SET DEFAULT**: same as ON DELETE NULL except that instead of setting the foreign key to NULL, it is set to the default value. As shown in Figure 8.2, this option can be specified as follows:

FOREIGN KEY (DNO) REFERENCES DEPARTMENT (DNUMBER) ON DELETE SET DEFAULT

- For example, in Figure 8.2, note that the DNO attribute in EMPLOYEE has an default value of 1. If department whose DNUMBER is 5 is deleted in the DEPARTMENT table, the DNO attribute will be set to 1 for the employees John Smith, Franklin Wong, Ramish Narayan & Joyce English.

- **ON UPDATE CASCADE**: if the primary key is updated, all corresponding foreign key values will be updated. As shown in Figure 8.2, this option is specified as follows:

FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT (DNUMBER) ON UPDATE CASCADE

- For example if *DNUMBER* in the **DEPARTMENT** table has been changed from 5 to 6, the same change will be applied to the **DEPT_LOCATIONS**, **EMPLOYEE**, **PROJECT** tables.

- **Giving Names to Constraints**

- Constraints can be given name as shown in Figure 8.2.
- All constraints within a schema must be given unique names.

- **Using CHECK to specify Constraints**

  - In addition to primary & foreign keys constraints, other table constraints can be specified using the **CHECK** constraint. .
  - For example, suppose that the **DEPARTMENT** table in Figure 8.1 has an additional attribute *DEPTCREATEDATE* which stores the date when the department was created.
  - At the end of the **CREATE TABLE** statement for the DEPARTMENT table, we could add the following **CHECK** clause to make sure that the manager's start date is later than the department creation date:

```
CHECK (DEPTCREATEDATE <  MGERSTARTDATE );
```

  - This constraint will be applied whenever a tuple is inserted or modified.

- **Schema Change Statements in SQL**

  - **The DROP command**

    - The **DROP** command is used to drop schema elements such on tables, domains and constraints.
    - This command can also be used to drop the schema itself using the **CASCADE** or **RESTRICT** options.
    - Using the **CASCADE** option, the schema and all its elements will be dropped:

```
DROP SCHEMA COMPANY CASCADE;
```

    - Using the **RESTRICT** option, the schema can be removed only if it has no elements, otherwise the **DROP** command will not be executed.

    - To drop a table within a schema:

```
DROP TABLE COMPANY.DEPENDENT CASCADE;
```

    - The **CASCADE** option above will cause all constraints referencing the table being dropped to be dropped as well.
    - If the **RESTRICT** option is used instead, the table is dropped only if it is not referenced in any constraint (for example, by foreign key constraints) in other tables.

    - The drop command can also be used to drop other schema elements like constraints and domains.

  - **ALTER Command**

    - The **ALTER** command is used to change schema elements.

- For tables, this command can be used to add or drop columns, change column definition and add or drop constraints.
- For example, to add an attribute called *JOB* to the **EMPLOYEE** table:

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

- By default each tuple will be assigned a NULL value for the new attribute. It can be changed later using the **UPDATE** command.
- A column can be dropped using either **CASCADE** or **RESTRICT**.
- **CASCADE** option will cause all constraints and views that reference the column to be dropped as well.
- **RESTRICT** option will drop the column only if it is not referenced by constraints or views.
- For example: the following command removes the attribute *Address* from the **EMPLOYEE** table:

```
ALTER TABLE COMPANY.EMPLOYEE DROP Address CASCADE;
```

- An attribute definition can be changed by dropping an existing default clause or defining a new default clause:

.

```
ALTER TABLE COMPANY.DEPARTMINT ALTER MGRSSN DROP DEFAULT;
ALTER TABLE COMPANY.DEPARTMINT ALTER MGRSSN SET DEFAULT '333445555';
```

- Also table constraints can be added or dropped. For example, to drop the constraint *EMPSUPERFK* in Figure 8.2:

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

- The **CASCADE** option causes any dependent constraints to also be dropped.
- For example, if you attempt to drop a **PRIMARY KEY**, you use **CASCADE** option to drop any corresponding foreign key constraints.

- Basic Queries in SQL

- SELECT-FROM-WHERE Structure

- Queries in SQL are expressed using the SELECT-FROM-WHERE block whose syntax is:

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

<attribute list > is a list of attribute names whose values are to be retrieved

<table list > is a list of relation names required to process the query.

<condition > is a conditional (Boolean) expression that identifies the tuples to be retrieved and uses one of the conditional operators =, <, <=, >, >= and <>

- **Examples**:

- **Query** 0: retrieve the birth date and address of the employee whose name is 'John B. Smith':

```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME='John'  AND MINIT='B' AND LNAME='Smith';
```

- The result is shown in Figure 8.3(a).

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

```
SELECT FNAME , LNAME , ADDRESS
FROM  EMPLOYEE , DEPARTMENT
 WHERE DNAME = 'RESEARCH'  AND  DNUMBER =DNO;
```

- The result is shown in Figure 8.3(b).

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

```
SELECT PNUMBER , DNUM , LNAME , ADDRESS ,BDATE
FROM PROJECT , DEPARTMENT ,EMPLOYEE
WHERE DNUM = DNUMBER AND MGRSSN =SSN
AND PLOCATION = 'Stafford';
```

- The result is shown in Figure 8.3(c).

- Ambiguous attribute Names, Aliasing & Tuple Variables

- Different relations can have the same attribute names where **DEPARTMENT** & **DEPT_LOCATIONS** have the same attribute *DNUMBER.*

- If an SQL query references attributes with the same names, the attributes must be qualified to resolve ambiguity.

- An attribute can be qualified by prefixing it with the relation name and separating the two by a period.

- **Example** : For all departments, list the department name, department number, manager SSN and department location.

```
SELECT  DNAME , DEPARTMENT.DNUMBER , MGRSSN,DLOCATION
FROM DEPARTMENT , DEPT_LOCATIONS
WHERE DEPARTMENT.DNUMBER= DEPT_LOCATIONS.DNUMBER;
```

- Aliases (alternative names) are used to give relation names short names. Using aliases, the above query can be expressed as follows:

```
SELECT A.DNAME, A.DNUMBE, A.MGRSSN, B.LOCATION
FROM DEPATMENT AS A , DEPT_LOCATIONS AS B
WHERE A.DNUMBER = B.DNUMBER;
```

- The **AS** keyword can be omitted.

- If is also possible to rename relation attributes within the query in SQL by giving them aliases as follows:

```
EMPLOYEE AS E(FN ,MI ,LN ,SSN ,BD, ADDR ,SEX ,SAL ,SSSN  ,DNO)
```

- In this case *FN* become an aliase for *FNAME* , *MI* for *MINIT* and so on.

- Unspecified Where Clause And Use Of The Asterisk

- A missing **WHERE** clause indicates no condition, therefore, all tuples will be selected:

```
SELECT DNO
FROM EMPLOYEE
```

- If more than one relation is specified in the **FROM** clause & the where clause is missing, the resulting table will contain the CROSS PROUDCT (all possible tuples) of the relations specified in the FROM clause .

- When the **FROM** clause contains more than one table, it is important to specify the join condition. For example, to retrieve the department name & location:

```
SELECT DNAME,DLOCATION
FROM DEPARTMENT AS A , DEPT_LOCATIONS AS B
WHERE A.DNUMBER = B.DNUMBER;
```

- To retrieve all attributes in the relation in the **FROM** clause, use the * operator :

```
SELECT *
FROM DEPT_LOCATIONS;
```

- The above query is equivalent to:

```
SELECT DNUMBER , DLOCATION
FROM DEPT_LOCATIONS;
```

- Complex SQL Queries

- Comparison Involving NULL & Three-Valued Logic

- In SQL, a NULL value has 3 different meanings:

  - **Unknown** value: a person has date of birth that is not known.

  - **Unavailable** or withheld value : is a person has home phone is not listed .

- **Not applicable** attribute: a value for the attribute LastCollegeDegree would be NULL for a person who has no college degree because if does not apply to this person .

- SQL does not distinguish between the different meanings of NULL because if is not possible to determine which of the meanings is intended.

- When a NULL value is involved in a compassion operation, the result is UNKNOWN.

- Therefore, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN.

- The truth values for these values when the logical connectives AND, OR and NOT Are used are shown in Table 8.1.

- **Tables as Sets in SQL**

- SQL treats a table as a multiset where duplicate tuples can appear more than once (unlike relational algebra , duplicate tuples are eliminated).

- One exception is an SQL table with a key which is restricted to being a set.

- To eliminate duplicate tuples from the result, the **DISTINCT** Keyword is used .

- To keep duplicate values, use **SELECT** or **SELECT ALL**.

- **Query 11**: Retrieve the salary of every employee .

```
SELECT ALL SALARY
FROM EMPLOYEE ;
```

Or

```
SELECT SALARY
FROM EMPLOYEE ;
```

- The result is shown in Figure 8.4 (a).

- To perform the same query but without duplicate values:

```
SELECT DISTINCT SALARY
FROM EMPLOYEE;
```

- The result is shown in Figure 8.4 (b).

- Set operations in SQL are set anion (**UNION**), Set difference (**EXCEPT**) & set intersection (**INTERSECT**).

- The relations resulting from these set operations are sets of tuples without duplicate values.

- The relations that the above set operations are applied to must be union compatible.

- **Query 12**: 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 projects:

```
(SELECT    DISTINCT  PNUMBER
 FROM       PROJECT,DEPARTMENT,EMPLOYEE
 WHERE      DNUM = DNUMBER AND
             MGRSSN=SSN AND LNAME = 'Smith')
UNION

(SELECT  DISTINCT   PNUMBER
 FROM    PROJECT,WORKS_ON, EMPLOYEE
 WHERE    PNUMBER = PNO AND ESSN=SSN AND LNAME = 'Smith');
```

- The operations UNION ALL, EXCEPT ALL & INTERSECT ALL are the same as UNION, EXCEPT & INTERSECT except that duplicate values are not eliminated.

- **Substring Pattern Matching & Arithmetic Operations**

- The **LIKE** operator is used for string pattern matching.

- It uses two reserved characters:
    - % to match 0 or more characters
    - _ to match 1 character

- **Query 13**: "Retrieve all employees whose address is in Houston, TX:

```
SELECT   FNAME,LNAME
FROM     EMPLOYEE
WHERE    ADDRESS LIKE ‘% Houston, TX%’;
```

- To retrieve all employees born in the 1950s:

```
SELECT  FNAME , LNAME
FROM    EMPLOYEE
WHERE   BDATE  LIKE  '__5________';
```

- To match the _ or % characters (that is, treat them as regular characters), they must be preceded by the ESCAPE character (‘\’).

- To match the apostrophe, precede it with another apostrophe:

STORE

NAME	LOCATION
Mr Jone's	123 Smith st.

```
SELECT    LOCATION
FROM      STORE
WHRE      NAME = ' Mr  Jone" s' ;
```

- Standard arithmetic operations (+, - , / ,*) can be used in queries to produce calculated fields .

- For example, to produce a calculated field called INCREASED_SAL which reflects a 10 percent raise for all employees working on the 'ProductX ' project:

```
 SELECT     FNAME , LNAME , 1.1*SALARY AS INCREASED_SAL
 FROM       EMPLOYEE,WORKS_ON, PROJECT
WHERE       SSN = ESSN  AND  PNO = PNUMBER AND  PNAME = 'ProductX ' ;
```

- Range values can be specified using the **BETWEEN** operator as follows:

- **Query 14**: Retrieve all employees in department 5 whose Salary is between $30,000 and $40,000:

```
SELECT   *
FROM    EMPLOYEE
WHERE  ( SALARY BTWEEN 30000 AND 40000) AND  DNO  = 5 ;
```

- Without using the **BETWEEN** operator, this query can be represented as follows:

```
SELECT *
FROM    EMPLOYEE
WHERE  (( SALARY >= 30000 AND  SALARY <=40000)
                 AND  DNO = 5 );
```

- Ordering of Query Results

- The **ORDER BY** clause is used to order tuples based on one or more attributes.
- By default , **ORDER BY** orders tables in ascending order.
-To order tuples in descending order, the keyword DESC is used with the **ORDER BY** clause .

- For example, consider the following table:

EMPLOYEE

EMP_ID	EMP_LNAME	EMP_FNAME	EMP_TITLE
001	Smith	Mike	Programmer
002	Robert	Hicken	Tester
003	Peter	Angle	Programmer
004	Peter	Jordan	CEO

- To retrieve employee last name, employee first name, employee title & order them by last name in ascending order & first name in descending order .

```
SELECT    EMP_LNAME  ,  EMP_FNAME  , EMP_TITLE
FROM       EMPLOYEE
ORDER  BY   EMP_LNAME ,  EMP_FNAME  DESC ;
```

- Result:

EMP_LNAME	EMP_FNAME	EMP_TITLE
Peter	Jordan	CEO
Peter	Angelo	Programmer
Robert	Hicken	Tester
Smith	Mike	Programmer

- Assume that the EMPLOYEE table has been updated by adding a new tuple (005,'Smith','Mike','Tester') as follows:

EMP_ID	EMP_LNAME	EMP_FNAME	EMP_TITLE
001	Smith	Mike	Programmer
002	Robert	Hicken	Tester
003	Peter	Angle	Programmer
004	Peter	Jordan	CEO
005	Smith	Mike	Tester

Show the result of the following query :

```
SELECT     EMP_LNAME, EMP_FNAME,EMP_TITLE
FROM        EMPLOYEE
ORDER BY  EMP_LNAME DESC , EMP_FNAME,EMP_TITLE DESC ;
```

- The keyword ASC can be used to specify ascending order but it is redundant because the default is ascending order .

- Nested Queries , Tuples & Set/Multiset Comparisons

- A nested query is a query within a query.
- This means that is complete **SELECT- FROM-WHERE** block can be specified in the where clause of another query called the outer query.

- **Query 12** above can be also expressed as:

```
SELECT   DISTINCT PNUMBER
FROM      PROJECT
WHERE    PNUMBER  IN

    (SELECT   PNUMBER
        FROM     PROJECT,DEPARTMENT
        WHERE   DNUM=DNUMBER  AND MGRSSN=SSN  AND
```

```
                                   LNAME='Smith')

                       OR PNUMBER IN

                       (SELECT   PNO
                                 FROM    WORKS_ON,EMPLOYEE
                                 WHERE   ESSN=SSN AND LNAME= 'Smith' ) ;
```

- Note above that the first nested query selects the project numbers of projects that have 'Smith' as a manager and the second nested query selects project numbers that have 'Smith' as a worker .

- Multiple attributes can be specified by enclosing them in parentheses as follows:

```
SELECT   DISTINCT   ESSN
FROM      WORKS_ON
WHERE    (PNO,HOURS)  IN

       (SELECT   PNO,HOURS
         FROM     WORKS_ON
         WHERE    ESSN='123456789' );
```

Question: Describe the intended meaning of the above query.

- In addition to the IN operator, the following operators can be used:

- =ANY: returns TRUE if the value v is equal to some value in the set V.
- >=ANY: returns TRUE if the value v is greater than or equal some value in the set V.
- <ANY: returns TRUE if the value v is less than some value in the set V.
- <=ANY: returns TRUE if the value v is less than or equal some value in the set V.
- <> ANY: returns TRUE if the value v is not equal any value in the set V.

- Instead of ANY, ALL can be used. For example the comparison condition (v > All V) returns TRUE if the value v is grater then all the values in the set V.

- **Example**: retrieve the name of employees whose salary is greater than the salary of all the employees in departments 5:

```
SELECT  LNAME, FNAME
FROM   EMPLOYEE
WHERE SALARY  > ALL
```

```
        (SELECT SALARY
          FROM EMPLOYEE
          WHERE DNO= 5);
```

- If the nested query has an attribute with the same name as the outer query, by default, a reference to this attribute in the nested query refers to the attribute in the nested query.
- To reference an attribute with the same name in the outer query, the attribute must be qualified as in the following query:

- **Query 15**: retrieve the name of each employee who has a dependent with the same first name and same sex as the employee:

```
SELECT  E.FNAME,E.LNAME
FROM   EMPLOYEE AS E
WHERE  E.SSN  IN

      (SELECT ESSN
        FROM DEPENDENT
       WHERE SSN=ESSN AND E.FNAME=DEPENDENT_NAME
                  AND E.SEX = SEX);
```

- **Correlated Nested Queries**

- When a nested query references an attribute in the outer query, the two queries are said to be correlated.

- In this case, the nested query is evaluated for each tuple in the outer query.

- **EXISTS Function**

- EXISTS function is used to check if the result of a correlated nested query is empty (contains no tuples).

- For example: the above query can be expressed as:

```
SELECT E.FNAME,E.LNAME
FROM EMPLOYE AS E
WHERE EXISTS    (SELECT *
                  FROM DEPENDENT
                  WHERE E.SSN=ESSN AND E.SEX=SEX
                           AND E.FNAME=DEPENDENT_NAME);
```

- The above query is evaluated as follows: for each Employee tuple, evaluate the nested query which retrieves all DEPENDENT tuples with the same social security number, sex and name as the EMPLOYEE tuple.
- In general, EXISTS(Q) returns TRUE if there is at least one tuple in the result of the nested query Q or FALSE otherwise, therefore NOT Exists (Q) returns TRUE if there are no tuples in the result of the nested query Q or FALSE otherwise.

- **Query 16**: retrieve the names of the employees who have no dependents.

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

- **Query 17**: list the names of mangers who have at least one dependent:

```
SELECT FNAME,LNAME
 FROM   EMPLYEE
 WHERE  EXISTS   ( SELECT *
                   FROM DEPENDENT
                   WHERE SSN=ESSN)
   AND EXISTS
                 (SELECT *
                  FROM DEPARTMENT
                   WHERE SSN=MGRSSN);
```

- **Query 18**: retrieve the name of each employee who works on all projects controlled by departments number 5:

```
SELECT FNAME,LNAME
FROM EMPLOYEE
WHERE NOT EXISTS
      ( (SELECT PNUMPER
          FROM PROJECT
            WHERE  DNUM=5)
        EXCEPT
       (SELECT  PNO
        FROM WORKS_ON
             WHERE SSN=ESSN));
```

- Explicit Sets & Renaming of attributes in SQL

- Rather than using a nested query, an explicit set of values can be used:

- **Query 19**: Retrieve the social security numbers of all employees who work on project numbers 1,2 or 3.

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

- Like relations, SQL allows attributes to be renamed.
- For example, the following query displays the attributes *EMPLOYEE_NAME* and *SUPERVISOR_NAME* instead of the attribute *E.LNAME & S.LMAME* respectively.

```
SELECT E.LNAME AS EMPLOYEE_NAME,S.LNAME AS SUPERVISOR_NAME
```

```
FROM    EMPLOYEE AS E,EMOLYEE AS S
WHERE  E.SUPERSSN=S.SSN;
```

- Joined Tables in SQL

- Tables related using primary and foreign keys can be joined in **FROM** clause using the keywords **JOIN** & **ON**.
- For Example, to retrieve the name and address of every employee who works for the 'Research' department:

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

- The above query contains an EQUIJOIN.
- A natural join can be specified using the keywords NATURAL JOIN.
- AS stated in Chapter 6, the natural join requires the join attributes to have the same name. If not, one of the attributes must be renamed. The above query can be represented using NATURAL JOIN as follows:

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

- The default type of join is inner join where a tuple included in the result only if matching tuple exists.
- left, right & full outer joins can be specified using **LEFT OUTER** join, **RIGHT OUTER** join & **FULL OUTER** join responsively.

- For Example: the following query retrieves the employee name and supervisor name of all employees include ones who have no supervisors:

```
SELECT E.LNAME AS EMPLOYEE_NAME,
            S.LNAME AS SUPERVISOR_NAME

FROM    (EMPLOYEE AS E LEFT OUTER JOIN
            EMPLOYEE AS S ON E.SUPERSSN=S.SSN);
```

- A relation resulting of joining two relations can be joined with another relation.
- **Query 2** (page 8) can also be expressed as follows:

```
SELECT    PNUMBER, DNUM, LNAME, ADDRESS, BDATE
FROM      ( ( PROJECT JOIN DEPARTMENT ON DNUM=DNUMBER)
JOIN EMPLOYEE ON MGRSSN=SSN)
WHERE PLOCATION='Stafford’;
```

- Aggregate Functions in SQL

- SQL supports the same aggregate functions supported in relational algebra.

- The functions are:

COUNT: returns the number of tuples or values as specified in a query.

SUM: returns the sum of numeric values.

MAX: returns the maximum value in a set of values.

MIN: returns the minimum value in a set of values.

AVG: returns the average value in a set of values.

- **Query 20**: find the sum of salaries of all employees, the maximum salary, the minimum salary & average salary:

```
SELECT  SUM ( SALARY ), MAX(SALARY ), MIN(SALARY), AVG(SALARY )
FROM  EMPLOYEE ;
```

- **Query 21**: same as query 20 except for the ' Research' department.

```
SELECT    SUM (SALARY), MAX(SALARY),MIN(SALARY),AVG(SALARY)
FROM      (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
WHERE     DNAME = ' Research' ;
```

- **Query 22**: retrieve the total number of employees in the company:

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

- **Query 23**: Retrieve the number of employees in the ' Research ' department :

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

- **Query 24**: Count the number of distinct salary values in the **EMPLOYEE** table :

```
SELECT    COUNT ( DISTINCT SALARY ) FROM EMPLOYEE;
```

- Note that tuples with null values in the salary column will not be included.

- Retrieve the names of all employees who have two or more dependents:

```
SELECT   LNAME , FNAME
FROM      EMPLOYEE
WHERE    (   SELECT COUNT ( * )
             FROM  DEPENDENT
             WHERE  SSN  = ESSN ) >=2;
```

- **Grouping: the GROUP BY & HAVING Clause**

- Tuples in relations can be divided into subgroups based on some attribute values. These attributes are called grouping attributes.

-Typically, one of the aggregate functions in the previous section is applied to each group to produce statistical results.

- The **GROUP BY** clause is used for this purpose.

- **Query 25**: for each department, retrieve the department number, # of employees in the department & their average salary.

```
SELECT     DNO, COUNT(*), AVG(SALARY)
FROM       EMPLOYEE
GROUP BY DNO ;
```

- In the above query ,tuples are partitioned into groups where each group has the same value for the grouping attribute *DNO*. The **COUNT** and **AVG** functions are then applied to each group .

- The result in shown in Figure 8.6(a).

- If NULL values exist in the grouping attribute, a separate group is created for all tuples with NULL values in the grouping attribute.

- **Query 26**: for each project, retrieve the project number, project name, and the number of employees who work on that project .

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

- In the above query, the grouping & functions are applied after joining the two tables.

- The relation resulting from joining the **PROJECT** & **WORKS_ON** relations is shown in Figure 8.6(b), therefore, the result of the previous query is:

PNUMBER	PNAME	COUNT(*)
1	ProductX	2
2	ProductY	2
3	ProductZ	2
10	Computerization	3
20	Reorganization	3
30	Newbenefits	3

- The **HAVING** clause is used to select certain groups that satisfy certain condition.

- For example , in **Query 26**, suppose we only want to list groups with more than two employees:

```
SELECT     PNUMBER ,PNAME,COUNT(*)
FROM        PROJECT , WORKS_ON
WHERE      PNUMBER = DNO
GROUP BY PNUMPER, PNAME
HAVING     COUNT (*) >2;
```

- **Query 27**: for each project, retrieve the project number, the project name, and the number of department 5 employees who work on this project.

```
SELECT      PNUMBER,PNAME,COUNT(*)
FROM        PROJECT , WORKS_ON,EMPLOYEE
WHERE       PNUMBER=PNO AND SSN = ESSN AND DNO =5
GROUP BY    PNUMBER, PNAME;
```

- When we have a **WHERE** clause & a **HAVING** clause, the rule is **WHERE** clause is executed first to select individual tuples then the **HAVING** clause in applied later to select individual groups of tuples.
- **Query 28**: for each dapartment that has more than 5 employees, retrieve the department number & the number of its employees whose salary is more than 40,000:

```
SELECT   DNUMBER ,COUNT(*)
FROM      DEPARTMENT , EMPLOYEE
WHERE    DNUMBER = DNO AND SALARY > 40000 AND
                        DNO IN   (    SELECT   DNO
                                      FROM      EMPLOYEE
                                      GROUP     BY    DNO
                                      HAVING   COUNT (*) >5);
GROUP  BY  DNUMBER ;
```

- **Summary of SQL Queries**

- SQL queries can consist of up to six classes but only the **SELECT** and **FROM** clauses are mandatory. The others are optional and enclosed between […]:

```
SELECT < ATTRIBUTE & FUNCTIONLIST >
FROM    < TABLE LIST >
[WHERE < condition >]
[GROUP BY < GROUPING ATTRIBUTE (S) >]
[HAVING < GROUP CONDITION >]
[ORDER BY < ATTRIBUTE LIST  >];
```

- **SELECT** clause: lists the attributes or functions to be retrieved.
- **FROM** clause: specifies all tables needed by the query.
- **GROUP BY** clause: specifies grouping attributes to produce groups or partitions.
- **HAVING** clause: specifies a condition on the groups being selected.
- **ORDER BY** clause: specifies an order for displaying query result.

- A query is evaluated as follows:

- **FROM** clause is evaluated to identify the tables & perform any join operation (if any).
- **WHERE** clause is evaluated against each tuple in the table in the FROM clause.
- **GROUP BY** clause is evaluated to produce groups based on the value of the grouping attribute(s).
- **HAVING** clause lists any conditions that will be used to filter the groups being produced by **GROUP BY** clause.
- Finally, **ORDER BY** clause is applied at the end to sort query result.

- Note that there are numerous ways to specify the same query in SQL. This flexibility has advantages and disadvantages.

Main advantages: Users can choose the technique they are most comfortable with.
Main disadvantages: Having numerous ways to specify the same query may confuse the user.

- Generally, it is preferable to write queries with as little nesting and ordering as possible.

- INSERT, DELETE, and UPDATE Statements in SQL

- INSERT Command

- Insert command is used to insert a single tuple to a relation.
- The values should be listed in the same order in which the corresponding attributes were specified in the **CREATE TABLE** command.
- For example, the following inserts a new employee:

```
INSERT   INTO   EMPLOYEE
VALUES  (‘ Richard’, ‘K’,   ‘Marini’, ‘653298653’, ‘1962-12-  30’,’98 Oak forest,
                Katy, TX’, ‘M’, 37000,   ‘987654321’, 4);
```

- A second form allows the user to specify by explicit attribute names that corresponds to the values specialized in the **INSERT** command.
- For example, to insert anew employee tuple for whom we only know *FNAME*, *LNAME*, *DNO* & *SSN*:

```
INSERT INTO   EPLOYEE (FNAME, LNAME, DNO, SSN) VALUES (‘Richard’, ‘Marini’, ‘4’,
‘653298653’);
```

- Note that values listed must correspond to all attributes with **NOT NULL** and no default values. Attributes not specified in the above query are set to their default or NULL.
- Integrity constraints should be enforced when applying the **INSERT** command. For example, the following **INSERT** command should be rejected because it violates referential integrity:

```
INSERT INTO    EPMLOYEE (FNAME, LANME, SSN, DNO)
VALUES             (‘Robert’, ‘Hatcher’, ‘980760540’, 2);
```

- A variation of the **INSERT** command uses a query to populate a new created table as follows:

```
CREATE TABLE    DEPTS_INFO
(DEPT_NAME          VARCHAR (50),
NO_OF_EMPS          INTEGER,
TOTAL_SAL           INTEGER);

INSERT INTO  DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,TOTAL_SAL)
SELECT  DNAME, COUNT (*), SUM (SALARAY)
FROM   (DEPARTMENT JOIN EMPLYEE ON DNUMBER=DNO);
GROUP BY  DNAME;
```

- Note that if the **DEPARTMENT** & **EMPLOYEE** tables are updated, the DEPTS_INFO table becomes outdated. If you want a table to always be updated, use views.

- **DELETE Command**

- **DELETE** command is used to delete a tuple from a relation.
- Tuples are deleted from only one table at a time. However, deletion may propagate to other relations by the **ON DELETE CASCADE** constraint for foreign keys.

- The **DELETE** command may use a **WHERE** clause to delete certain tuples, otherwise, all tuples will be deleted.
- For example to delete an employee whose last name is 'Brown':

```
DELETE   FROM    EMPLOYEE
 WHERE   LNAME = 'Brown';
```

- To delete employee with SSN = '123456789':

```
DELETE  FROM     EMPLOYEE
WHERE    SSN = '123456789';
```

- **Exercise**: Describe in English the following query:

```
DELETE  FROM    EMPLOYEE WHERE   DNO IN (SELECT    DNUMBER
                                         FROM       DEPARTMENT
                                         WHERE DNAME= ' Research');
```

- **UPDATE Command**

- **UPDATE** command is used to modify attribute values for one or more selected tuples.
- Certain tuples to be updated can be specified using the **WHERE** clause.
- Updating one relation may prorogate to other relations if the **ON UPDATE CASCADE** constraint is used for foreign key.
- For example, to change the location and controlling department number of project

number 10 to ‘Bellaire’ and 5 respectively:

```
UPDATE    PROJECT
SET         PLOCATION = ‘Bellaire’, DNUM = 5
WHERE     PNUMBER = 10;
```

- Several tuples can be modified with a simple **UPDATE** command:

```
UPDATE    EMPLOYEE  SALARY = SALARY*1.1
WHERE     DNO IN (SELECT     DNUMBER
                  FROM        DEPARTMENT
                  WHERE      DNAME = ‘Research’);
```

- Specifying Constraints as Assertions and Triggers

- Assertions

- Other than the primary key, foreign key, NOT NULL, CHECK, and DEFAULT constraints, SQL allows general constraints via declarative assertions.

- A declarative assertion can be created using the **CREATE ASSERTION** statement of the DDL.
- For exmaple, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for, we can write the following assertion:

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

- Note above that the constraint name SALARY_CONSTRAINT is followed by the keyword **CHECK**, which is followed by a condition in parentheses that must hold true on every database state for the assertion to be satisfied.

- Whenever a tuple causes the condition to evaluate FALSE, the constraint is violated.

- Also note that the constraint is enforced when a tuple is inserted or modified.

- Triggers

- Triggers will be covered in the lab activity of this course.

- Views (Virtual Tables) in SQL

- A view is a single table that is derived from other tables called the base tables.
- Unlike base tables, tuples of a view are not physically stored and therefore, a view is considered a virtual table.
- A view is used for two main reasons:

1) Frequently used queries, especially those using the join operation, can be represented as a view, to present the information to the user as as single table.
2) Security - use a view to restrict access to your table data by writing a view and include in this view, the desired columns and rows.

- A view is expressed as a query as shown in the following examples.

- Examples:

- Example 1:

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

- **Question**: Draw the view WORKS_ON1 showing the columns and rows.

- Example 2:

```
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;
```

- **Question**: Draw the view DEPT_INFO showing the columns and rows.

- After a view has been created, it can be queried the same way as base tables:

```
QV1:   SELECT Fname, Lname
       FROM   WORKS_ON1
       WHERE Pname='ProductX'
```

When a view is not needed anymore, it can be disposed using the **DROP VIEW** command as follows:

```
DROP VIEW WORKS_ON1
```

DATABASE APPLICATION DEVELOPMENT

A relational DBMS supports an SQL interface, and users can directly enter SQL commands. This simple approach is fine as long as the task at hand can be accomplished entirely with SQL commands. In practice, we may want to integrate a database application with a nice graphical user interface, or we may want to integrate with other existing applications.

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBMS to interact with it. Once a connection is established, SQL commands can be used to insert, delete, and modify data. SQL queries can be used to retrieve desired data, but we need to bridge an important difference in how a database system sees data and how an application program in a language like Java or C sees data: The result of a database query is a set (or multiset) or records, but Java has no set or multiset data type. This mismatch is resolved through additional SQL constructs that allow applications to obtain a handle on a collection and iterate over the records one at a time.

Embedded SQL allows us to access data using static SQL queries in application code. With **Dynamic SQL**, we can create the queries at run-time. **Cursors** bridge the gap between set-valued query answers and programming languages that do not support set-values.

The emergence of Java as a popular application development language, especially for Internet applications, has made accessing a DBMS from Java code a particularly important topic. **JDBC**, a programming interface that allows us to execute SQL queries from a Java program and use the results in the Java program. JDBC provides greater portability than Embedded SQL or Dynamic SQL, and offers the ability to connect to several DBMSs without recompiling the code.

SQLJ, which does the same for static SQL queries, but is easier to program in than Java, with JDBC. Often, it is useful to execute application code at the database server, rather than just retrieve data and execute application logic in a separate process. **Stored procedures**, which enable application logic to be stored and executed at the database server.

Embedded SQL

- SQL commands can be executed from within a program in a host language such as C or Java.
- The use of SQL commands within a host language program is called **Embedded SQL.** SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language.
- Also, any host language variables used to pass arguments into an SQL command must be declared in SQL.

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION.

Example

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long csid;
```

```
        short crating;
        float cage;
        EXEC SQL END DECLARE SECTION
```

Error Handling

The SQL-92 standard has two special variables for reporting errors, SQLCODE and SQLSTATE.

SQLCODE is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes.

SQLSTATE associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported.

One of these two variables must be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char [6], that is, a character string five characters long.

Embedding SQL Statements

SQL statements must be prefixed by EXEC SQL. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

```
        EXEC SQL
        INSERT INTO Sailors VALUES (:c_sname, :csid, :crating, :cage);
```

The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

```
        EXEC SQL WHENEVER [SQLERROR I NOT FOUND] [ CONTINUE I GOTO st'mt ]
```

The value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt*, which is presumably responsible for error and exception handling. Control is also transferred to *stmt* if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

Cursors (Why cursors are needed?)

A major problem in embedding SQL statements in a host language like C is that an ***impedance mismatch*** occurs because SQL operates on *set* of records, whereas languages like C do not cleanly support a set-of-records abstraction.

The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation. This mechanism is called a **cursor**. Declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next *n,* to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Example:

```
        EXEC SQL SELECT S.sname, S.age
                INTO :c_sname, :c_age
```

```
FROM Sailors S
WHERE S.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables *c_sname* and *c_age*. Therefore, we do not need a cursor to embed this query in a host language program.
But in the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable *c_minrating?*

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor sinfo is defined. Cursors are used to process individual rows returned by queries.

Basic Cursor Definition and Usage

Cursor is a control structure that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement.
Cursors are used to process individual rows returned by queries. Cursors enable manipulation of whole result sets at once. It enables the rows in a result set to be processed sequentially and perform complex logic on a row by row basis.

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
[WITH HOLD]
FOR some query
[ORDER BY order-item-list]
[FOR READ ONLY I FOR UPDATE]
```

Example

```
DECLARE sinfo CURSOR FOR
        SELECT S.sname, S.age
        FROM Sailors S
        WHERE S.rating > :c_minrating;
```

Properties of Cursors

- A cursor can be declared to be a read-only cursor (**FOR READ ONLY**) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (**FOR UPDATE**).
- If it is updatable, simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned.
- A cursor is updatable by default unless it is a scrollable or insensitive cursor, in which case it is read-only by default.
- If the keyword **SCROLL** is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed.
- If the keyword **INSENSITIVE** is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows.
- Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior.
- A holdable cursor is specified using the **WITH HOLD** clause, and is not closed when the transaction is committed.
- The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.
- In general, order in which FETCH commands retrieve rows is unspecified, but the optional ORDER BY clause can be used to specify a sort order. An order-item is a column name, optionally followed by one of the keywords ASC or DESC. The default is ASC
- Every column mentioned in the ORDER BY clause must also appear in the select-list of the query associated with the cursor;

Open the cursor:

OPEN sinfo:

A cursor points to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row.

Fetch the cursor:

The FETCH command can be used to read the first row of cursor sinfo into host language variables:

FETCH sinfo INTO :csname, :cage;

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement all the rows computed by the query can be read, one row at a time.

The cursor SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows if the FETCH statement positions the cursor after the last row.

Close the cursor:

When done with a cursor, it can be closed:

CLOSE sinfo;

Dynamic SQL

Dynamic SQL is an enhanced form of Structured Query Language (SQL) that, unlike standard (or static) SQL, facilitates the automatic generation and execution of program statements. This can be helpful when it is necessary to write code that can adjust to varying databases, conditions, or servers. It also makes it easier to automate tasks that are repeated many times.

Dynamic SQL statements are stored as strings of characters that are entered when the program runs. They can be entered by the programmer or generated by the program itself, but unlike static SQL statements, they are not embedded in the source program. Also in contrast to static SQL statements, dynamic SQL statements can change from one execution to the next.

Dynamic SQL statements can be written by people with comparatively little programming experience, because the program does most of the actual generation of the code. A potential problem is reduced performance (increased processing time) if there is too much dynamic SQL running at any given time.

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :csqlstring;
EXEC SQL EXECUTE readytogo;
```

AN INTRODUCTION TO JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. A DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. Even though the source code can be compiled to work with different DBMSs, the final executable works only with one specific DBMS.

Open DataBase Connectivity (ODBC) and Java DataBase Connectivity(JDBC), also enable the integration of SQL with a general-purpose programming language. Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API).

In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs 'Without recompilation. Thus, while Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are **DBMS-independent** at the source code level and at the level of the executable.

In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously.

All direct interaction with a specific DBMS happens through a DBMS-specific driver. A **driver** is a software program that translates the ODBC or JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a **driver manager**.

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source. There is no limit on the number of open connections, and an application can have several open connections to different data sources. Each connection has transaction semantics; that is, changes from one connection are visible to other connections only after the connection has committed its changes.

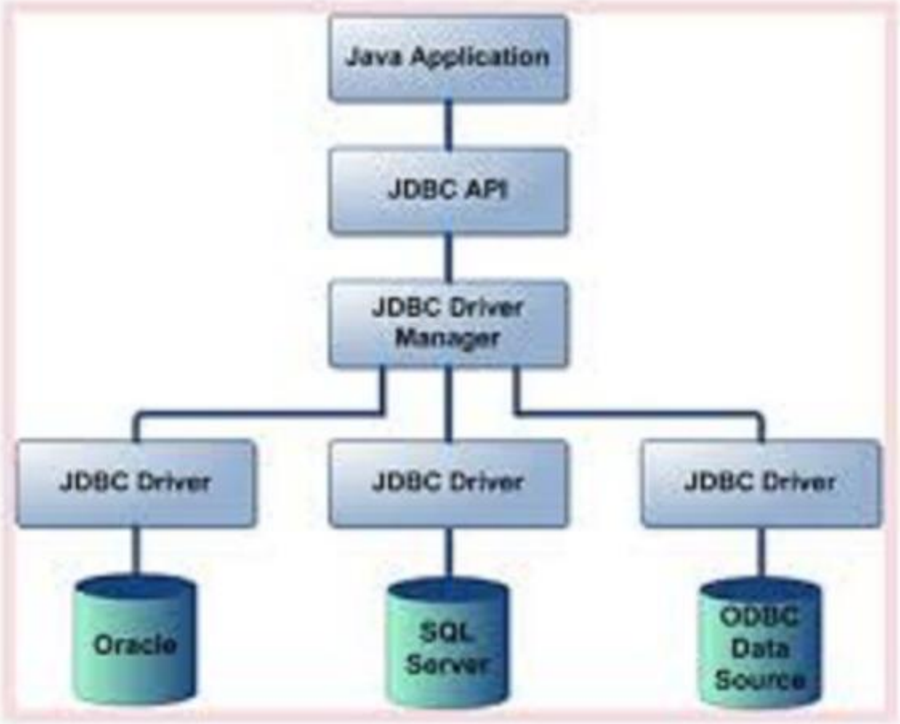
While a connection is open, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.

Architecture

The architecture of JDBC has four main components:

- **the application**: The application initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API
- **the driver manager:** The primary goal of driver manager is to load JDBC drivers and pass JDBC function calls from the application to the correct driver. It also handles JDBC initialization and information calls from applications and can log all function calls. Performs some error checking also.

- **several data source specific drivers:** The driver establishes connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard.
- **the corresponding data sources**: The data source processes commands from the driver and returns the results.



Four types of Drivers in JDBC

Drivers in JDBC are classified into four types depending on the architectural relationship between the application and the data source:

Type I Bridges: This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge. Bridges have the advantage that it is easy to piggyback the application onto an existing installation, and no new drivers have to be installed. The increased number of layers between data source and application affects performance. In addition, the user is limited to the functionality that the ODBC driver
supports.

Type II Direct Translation to the Native API via Non-Java Driver: This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is usually ,written using a combination of C++ and Java; it is dynamically linked and specific to the data source. This architecture performs significantly.One disadvantage is that the database driver that implements the API needs to be installed on each computer that runs the application.

Type III Network Bridges: The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (ie., the network bridge) is not DBMS-specific. The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server. The middleware
server can then use a Type II JDBC driver to connect to the data source.

Type IV-Direct Translation to the Native API via Java Driver: Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets. In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system. This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

JDBC CLASSES AND INTERFACES

- JDBC is a collection of Java classes and interfaces that enables database access from programs written in the java language.
- It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling.
- The classes and interfaces are part of the java. sql package.

JDBC Driver Management

- In JDBC, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers.
- The Drivermanager class has methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.
- The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished using static method forName in the Class class returns the Java class as specified in the argument string and executes its static constructor.
- The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

```
Class.forName("oracle/jdbc.driver.OracleDriver");
```

- After registering the driver, we connect to the data source.

Connections

- A session with a data source is started through creation of a Connection object;
- A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source.
- Connections are specified through a **JDBC** URL, a URL that uses the jdbc protocol. Such a URL has the form

jdbc:<subprotocol>:<otherParameters>

The code example shown below establishes a connection to an Oracle database assuming that the strings userld and password are set to valid values.

```
String url = “jdbc:oracle:www.bookstore.com:3083”
Connection connection;
try {
Connection connection = DriverManager.getConnection(urI, userId,password);
}
catch(SQLException
 excpt) {
        System.out.println(excpt.getMessageO);
        return;
}
```

In JDBC, connections can have different properties.

- to set the autocommit mode (Connection. setAutoCommit) and to retrieve the current autocommit mode (getAutoCommit).
- public int getTransactionIsolation() and public void setTransactionlsolation(int 1): get and set the current level of isolation for transactions handled in the current connection.
- public boolean getReadOnly() and public void setReadOnly(boolean readOnly): allow the user to specify whether the transactions executed through this connection are read only.
- public boolean isClosed() Checks whether the current connection has already been closed.

Executing SQL Statements

JDBC supports three different ways of executing statements:

Statement, PreparedStatement, and CallableStatement

The Statement class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.

The PreparedStatement class dynamically generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object is created.

Example:

```
 String sql = "INSERT INTO Books VALUES('?, 7, '?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
/ / assume that isbn, title, etc. are Java variables that contain the values to be inserted
pstmt.clearParameters() ;
pstmt.setString(l, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);
int numRows = pstmt.executeUpdate();
```

The SQL query specifies the query string, but uses '?' for the values of the parameters, which are set later using methods setString, setFloat, and setlnt. The '?' placeholders can be used anywhere in SQL statements where they can be replaced with a value.
clearParameters() before setting parameter values in order to remove any old data.

The query string is submitted to the data source using: executeUpdate and executeQuery commands.

- **ExecuteUpdate** command is used if SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The executeUpdate method returns an integer indicating the number of rows the SQL statement modified; it returns 0 for successful execution without modifying any rows.

- The **executeQuery** method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object.
- The **execute** method is more general than executeQuery and executeUpdate;

ResultSets

- The statement executeQuery returns a, ResultSet object, which is similar to a cursor.
- ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions.
- The *java.sql.ResultSet* interface represents the result set of a database query.
- The ResultSet object allows us to read one row of the output of the query at a time.
- Initially, the ResultSet is positioned before the first row, and we have to retrieve the first row with an explicit call to the next() method. The next method returns false if there are no more rows in the query answer, and true otherwise.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
/ / rs is now a cursor first call to rs.next() moves to the first record
/ / rs.next() moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
        / / process the data
}
```

• previous() moves back one row.
• absolute (int num) moves to the row with the specified number.
• relative (int num) moves forward or backward (if num is negative) relative to the current position. relative (-1) has the same effect as previous.
• first() moves to the first row, and last() moves to the last row.

Viewing a Result Set: The ResultSet interface contains methods for getting the data of the current row. There is a get method for each of the possible data types, and each get method has two versions –Example

public int getInt(String columnName) throws SQLException

Returns the int in the current row in the column named columnName

public int getInt(int columnIndex) throws SQLException

Returns the int in the current row in the specified column index.

Updating a Result Set: The ResultSet interface contains a collection of update methods for updating the data of a result set.

public void updateString(int columnIndex, String s) throws SQLException

Changes the String in the specified column to the value of s.

public void updateString(String columnName, String s) throws SQLException

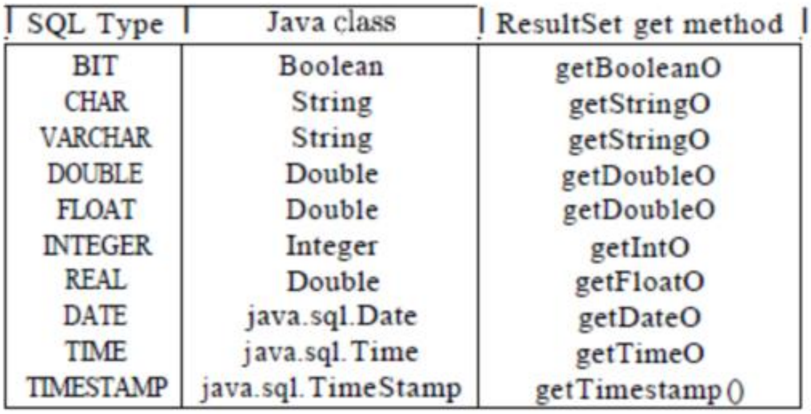
the column is specified by its name instead of its index.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

public void updateRow() : Updates the current row by updating the corresponding row in the database.
public void deleteRow(): Deletes the current row from the database
public void refreshRow(): Refreshes the data in the result set to reflect any recent changes in the database.
public void cancelRowUpdates(): Cancels any updates made on the current row.
public void insertRow(): Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

Matching Java and SQL Data Types

JDBC provides special data types and specifies their relationship to corresponding SQL data types. The table below shows accessor methods in a ResultSet object to retrieve values from the current row of the query result referenced by the ResultSet object for the most common SQL data types.



SQL Type	Java class	ResultSet get method
BIT	Boolean	getBooleanO
CHAR	String	getStringO
VARCHAR	String	getStringO
DOUBLE	Double	getDoubleO
FLOAT	Double	getDoubleO
INTEGER	Integer	getIntO
REAL	Double	getFloatO
DATE	java.sql.Date	getDateO
TIME	java.sql.Time	getTimeO
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Exceptions and Warnings

Most of the methods in java.sql can throw an exception of the type SQLException if an error occurs. The information includes SQLState, a string that describes the error. In addition to the standard getMessage() method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

public String getSQLState() returns an SQLState identifier based on the SQL:1999 specification.
public int getErrorCode () retrieves a vendor-specific error code.
public SQLException getNextException() gets the next exception in a chain of exceptions associated with the current SQLException object.
An **SQLWarning** is a subclass of SQLException. Warnings are not as severe as errors and the program can usually proceed without special handling of warnings. Warnings are not thrown like other exceptions, and they are not caught as part of the try"-catch block around a java. sql statement.

getWarnings() method with which we can retrieve SQL warnings if they exist. Duplicate retrieval of warnings can be avoided through **clearWarnings**().

Examining Database Metadata

We can use the DatabaseMetaData object to obtain information about the database system itself, as well as information from the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
DatabaseMetaData md = con.getMetaData():
System.out.println("Driver Information:");
```

STORED PROCEDURES

A **stored procedure** is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary.

A stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server.

Uses for stored procedures include data-validation or access-control mechanisms. To save time and memory, extensive or complex processing that requires execution of several SQL statements can be saved into stored procedures, and all applications call the procedures. One can use nested stored procedures by executing one stored procedure from within another.

Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easily.

Creating a Simple Stored Procedure

CREATE PROCEDURE *procedure_name* AS *sql_statement*;

Example1:

```
CREATE PROCEDURE ShowNumberOfOrders
                    SELECT C.cid, C.cname, COUNT(*)
                    FROM Customers C, Orders a
                    WHERE C.cid = O.cid
                    GROUP BY C.cid, C.cname
```

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT.

IN parameters are arguments to' the stored procedure.

OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process.

INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values.

Example2:
```
CREATE PROCEDURE Addlnventory (IN book_isbn CHAR(10),IN addedQty INTEGER)
                          UPDATE Books
```

```
SET qty_in_stock = qtyjn_stock + addedQty
WHERE bookjsbn = isbn
```

Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

CALL storedProcedureName(argumentl, argument2, ... , argumentN);

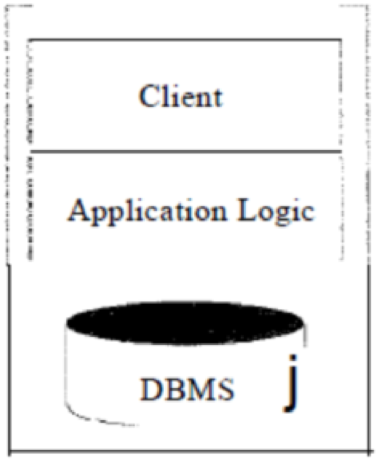
In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure Add Inventory would be called as follows:

EXEC SQL CALL AddInventory(:isbn,:qty);

THE THREE-TIER APPLICATION ARCHITECTURE

Single-Tier Architecture

Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure below. The application typically ran on a mainframe, and users accessed it through *dumb terminals* that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator.

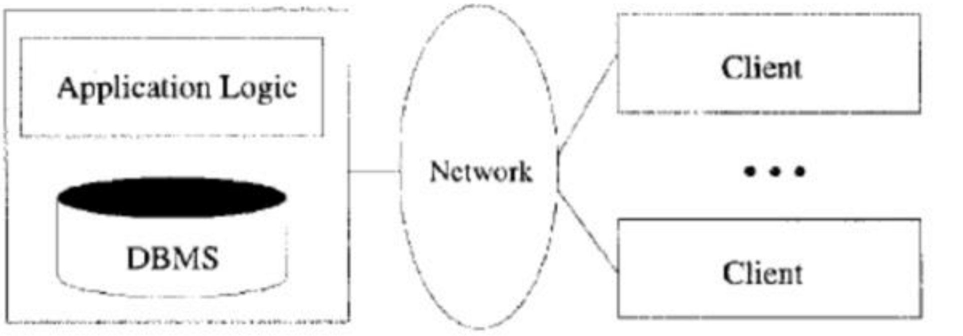


drawback: Users expect graphical interfaces that require much more computational power than simple dumb terminals. Single-tier architectures do not scale to thousands of users. The commoditization of the PC and the availability of cheap client computers led to the development of the two-tier architecture.

Two-tier architectures (Client-Server Architectures)

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol.

Client server architecture, the client implements just the graphical user interface, and the server implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustrated in below fig.

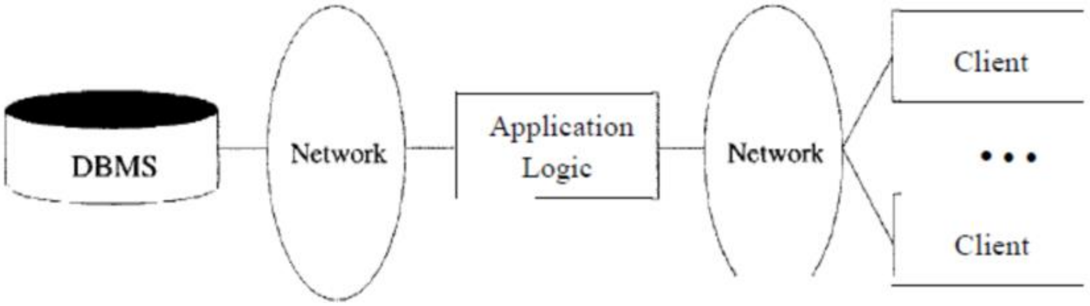


Other divisions are possible, such as more powerful clients that implement both user interface and business logic, or clients that implement user interface and part of the business logic, with the remaining part being implemented at the server level; such clients are often called **thick** clients, and this architecture is illustrated in fig.
the client side no longer have dumb terminals and require computers that run sophisticated presentation code (and possibly, application logic).
The thick-client model has several disadvantages when compared to the thin client model. First, there is no central place to update and maintain the business logic, since the application code runs at many client sites. Second, a large amount of trust is required between the server and the clients.
A third it does not scale with the number of clients; it typically cannot handle more than a few hundred clients. The application logic at the client issues SQL queries to the server and the server returns the query result to the client, where further processing takes place. Large query results might be transferred between client and server.

Three Tier Architectures

The three-tier architecture goes one step further, and also separates application logic from data management:

- Presentation Tier: Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- Middle Tier: The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general-purpose language such as *C*++ or Java.
- Data Management Tier: Data-intensive Web applications involve DBMSs, which are the subject of this book.



Advantages of the Three-Tier Architecture
The three-tier architecture has the following advantages:

- Heterogeneous Systems: Applications can utilize the strengths of different platforms and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.
- Thin Clients: Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
- Integrated Data Access: In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.
- Scalability to Many Clients: Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately.
- Software Development Benefits: By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages. The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APls. Therefore, each application tier can be built out of reusable components that can be individually developed, debugged, and tested.

Questions

1. List the data types that are allowed for SQL attributes.
2. In SQL which common is used for table creation? Explain how constraints are specified in SQL during table creation with suitable example. (Jan 2018)
3. Explain with suitable example, how you can retrieve information from multiple tables. (Jan-2017)
4. What is SQL? What are two major categories (DDL, DML) of SQL commands?
5. Discuss different types of constraints along with syntax that can be specified while creating relation in SQL.
6. Explain select-from-where clause with example.
7. Explain referential integrity constraints and its implementation in SQL.
8. Explain following commands with syntax example

   -CREATE
   -ALTER
   -DROP
   -SELECT
   -DELETE
   -UPDATE
   -GROUP BY and HAVING
   -EXISTS and NOT EXISTS
   -INSERT
   -IS NULL
   -JOIN

9. Write a short note on the following w.r.t SQL

   a. Aggregate functions in SQL

   b. Substring pattern matching