DATABASE MANAGEMENT SYSTEM

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2016 -2017)

SEMESTER – V

Subject Code 15CS53	IA Marks 20
Number of Lecture Hours/Week 4	Exam Marks 80
Total Number of Lecture Hours 50	Exam Hours 03

CREDITS – 04

Course objectives: This course will enable students to

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

Module – 1 **Teaching Hours 10**

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 Modeling using Entities and Relationships:** Entity types, Entity sets, attributes, roles, and structural constraints, Weak entity types, ER diagrams, examples, Specialization and Generalization.

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

Module – 2 **Teaching Hours 10**

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. **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.

Textbook 1: Ch4.1 to 4.5, 2.1 to 2.3, 6.1 to 6.5, 8.1; Textbook 2: 1.15

Module – 3 **Teaching Hours 10**

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. **Internet Applications:** The three-Tier application architecture, The presentation layer, The Middle Tier

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

Module – 4 **Teaching Hours 10**

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

Module – 5 **Teaching Hours 10**

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.

Introduction to Database Recovery Protocols: Recovery Concepts, NO-UNDO/REDO recovery based on Deferred update, Recovery techniques based on immediate update, Shadow paging, Database backup and recovery from catastrophic failures

Textbook 1: 20.1 to 20.6, 21.1 to 21.7, 22.1 to 22.4, 22.7.

Course outcomes: The students should be able to:

- Identify, analyze and define database objects, enforce integrity constraints on a Database using RDBMS.
- Use Structured Query Language (SQL) for database manipulation.
- Design and build simple database systems
- Develop application to interact with databases.

Question paper pattern:

The question paper will have TEN questions.

There will be TWO questions from each module.

Each question will have questions covering all the topics under a module.

The students will have to answer FIVE full questions, selecting ONE full question from each module.

Text Books:

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

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

Reference Books:

1. SilberschatzKorth and Sudharshan, Database System Concepts, 6th Edition, Mc- GrawHill, 2013.

MODULE-1

Introduction to Database

Introduction

Database is a collection of related data. Database management system is software designed to assist the maintenance and utilization of large scale collection of data. DBMS came into existence in 1960 by Charles. Integrated data store which is also called as the first general purpose DBMS. Again in 1960 IBM brought IMS-Information management system. In 1970 Edgor Codd at IBM came with new database called RDBMS. In 1980 then came SQL Architecture- Structure Query Language. In 1980 to 1990 there were advances in DBMS e.g. DB2, ORACLE.

Data

- Data is raw fact or figures or entity.
- When activities in the organization takes place, the effect of these activities need to be recorded which is known as Data.

Information

- Processed data is called information
- The purpose of data processing is to generate the information required for carrying out the business activities.

In general data management consists of following tasks

- Data capture: Which is the task associated with gathering the data as and when they originate.
- Data classification: Captured data has to be classified based on the nature and intended usage.
- Data storage: The segregated data has to be stored properly.
- Data arranging: It is very important to arrange the data properly
- Data retrieval: Data will be required frequently for further processing,

Hence it is very important to create some indexes so that data can be retrieved easily.

• Data maintenance: Maintenance is the task concerned with keeping the data upto-date.

• Data Verification: Before storing the data it must be verified for any error.

• Data Coding: Data will be coded for easy reference.

• Data Editing: Editing means re-arranging the data or modifying the data for presentation.

• Data transcription: This is the activity where the data is converted from one form into another.

• Data transmission: This is a function where data is forwarded to the place where it would be used further.

Metadata (**meta data**, or sometimes **meta information**) is "data about data", of any sort in any media. An item of metadata may describe a collection of data including multiple content items and hierarchical levels, for example a database schema. In data processing, metadata is definitional data that provides information about or documentation of other data managed within an application or environment. The term should be used with caution as all data is about something, and is therefore metadata.

Database

• Database may be defined in simple terms as a collection of data

• A database is a collection of related data.

• The database can be of any size and of varying complexity.

• A database may be generated and maintained manually or it may be computerized.

Database Management System

• A Database Management System (DBMS) is a collection of program that enables user to create and maintain a database.

• The DBMS is hence a general purpose software system that facilitates the process of defining constructing and manipulating database for various applications.

Characteristics of DBMS

• To incorporate the requirements of the organization, system should be designed for easy maintenance.

• Information systems should allow interactive access to data to obtain new information without writing fresh programs.

• System should be designed to co-relate different data to meet new requirements.

• An independent central repository, which gives information and meaning of available data is required.

• Integrated database will help in understanding the inter-relationships between data stored in different applications.

• The stored data should be made available for access by different users simultaneously.

• Automatic recovery feature has to be provided to overcome the problems with processing system failure.

DBMS Utilities

• A data loading utility:

Which allows easy loading of data from the external format without writing programs.

• A backup utility:

Which allows to make copies of the database periodically to help in cases of crashes and disasters.

• Recovery utility:

Which allows to reconstruct the correct state of database from the backup and history of transactions.

• Monitoring tools:

Which monitors the performance so that internal schema can be changed and database access can be optimized.

• File organization:

Which allows restructuring the data from one type to another?

Difference between File system & DBMS

File System

1. File system is a collection of data. Any management with the file system, user has to write the procedures
2. File system gives the details of the data representation and Storage of data.
3. In File system storing and retrieving of data cannot be done efficiently.
4. Concurrent access to the data in the file system has many problems like
a. Reading the file while other deleting some information, updating some information
5. File system doesn't provide crash recovery mechanism.
Eg. While we are entering some data into the file if System crashes then content of the file is lost.
6. Protecting a file under file system is very difficult.

DBMS

1. DBMS is a collection of data and user is not required to write the procedures for managing the database.
2. DBMS provides an abstract view of data that hides the details.
3. DBMS is efficient to use since there are wide varieties of sophisticated techniques to store and retrieve the data.
4. DBMS takes care of Concurrent access using some form of locking.
5. DBMS has crash recovery mechanism, DBMS protects user from the effects of system failures.
6. DBMS has a good protection mechanism.

DBMS = Database Management System

RDBMS = Relational Database Management System

A database management system is, well, a system used to manage databases.

A relational database management system is a database management system used to manage relational databases. A relational database is one where tables of data can have relationships based on primary and foreign keys.

Advantages of DBMS.

Due to its centralized nature, the database system can overcome the disadvantages of the file system-based system

1. **Data independency**:

Application program should not be exposed to details of data representation and storage DBMS provides the abstract view that hides these details.

2. **Efficient data access**.:

DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently.

3. **Data integrity and security**:

Data is accessed through DBMS, it can enforce integrity constraints.

E.g.: Inserting salary information for an employee.

4. **Data Administration**:

When users share data, centralizing the data is an important task, Experience professionals can minimize data redundancy and perform fine tuning which reduces retrieval time.

5. **Concurrent access and Crash recovery:**

DBMS schedules concurrent access to the data. DBMS protects user from the effects of system failure.

6. **Reduced application development time**.

DBMS supports important functions that are common to many applications.

Functions of DBMS

• Data Definition: The DBMS provides functions to define the structure of the data in the application. These include defining and modifying the record structure, the type and size of fields and the various constraints to be satisfied by the data in each field.

• Data Manipulation: Once the data structure is defined, data needs to be inserted, modified or deleted. These functions which perform these operations are part of DBMS. These functions can handle plashud and unplashud data manipulation needs. Plashud queries are those which form part of the application. Unplashud queries are ad-hoc queries which performed on a need basis.

• Data Security & Integrity: The DBMS contains modules which handle the security and integrity of data in the application.

• Data Recovery and Concurrency: Recovery of the data after system failure and concurrent access of records by multiple users is also handled by DBMS.

• Data Dictionary Maintenance: Maintaining the data dictionary which contains the data definition of the application is also one of the functions of DBMS.

• Performance: Optimizing the performance of the queries is one of the important functions of DBMS.

Role of Database Administrator.

Typically, there are three types of users for a DBMS:

1. The END User who uses the application. Ultimately he is the one who actually puts the data into the system into use in business. This user need not know anything about the organization of data in the physical level.

2. The Application Programmer who develops the application programs. He/She has more knowledge about the data and its structure. He/she can manipulate the data using his/her programs. He/she also need not have access and knowledge of the complete data in the system.

3. The Data base Administrator (DBA) who is like the super-user of the system.

The role of DBA is very important and is defined by the following functions.

• Defining the schema: The DBA defines the schema which contains the structure of the data in the application. The DBA determines what data needs to be present in the system and how this data has to be presented and organized.

• Liaising with users: The DBA needs to interact continuously with the users to understand the data in the system and its use. • Defining Security & Integrity checks: The DBA finds about the access restrictions to be defined and defines security checks accordingly. Data

Integrity checks are defined by the DBA.

• Defining Backup/Recovery Procedures: The DBA also defines procedures for backup and recovery. Defining backup procedure includes specifying what data is to be backed up, the periodicity of taking backups and also the medium and storage place to backup data.

• Monitoring performance: The DBA has to continuously monitor the performance of the queries and take the measures to optimize all the queries in the application.

Simplified Database System Environment

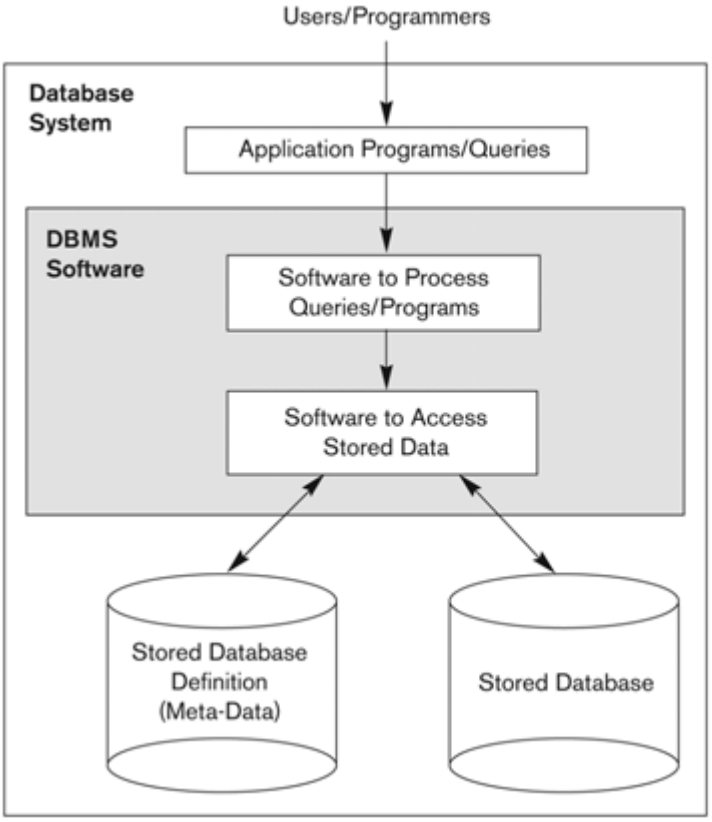


Figure 1.1
A simplified database system environment

A database management system (DBMS) is a collection of programs that enables users to create and maintain database. The DBMS is a general purpose software system that facilitates the process of defining, constructing, manipulating and sharing databases among various users and applications.

Defining a database specifying the database involves specifying the data types, constraints and structures of the data to be stored in the database. The descriptive information is also stored in the database in the form database catalog or dictionary; it is called meta-data.

Manipulating the data includes the querrying the database to retrieve the specific data.

An application program accesses the database by sending the qurries or requests for data to DBMS.

The important function provided by the DBMS includes protecting the database and maintain the database.

Example of a Database (with a Conceptual Data Model)

• **Mini-world for the example:**

Part of a UNIVERSITY environment.

• **Some mini-world *entities:***

STUDENTs

COURSEs

SECTIONs (of COURSEs)

(academic) DEPARTMENTs

INSTRUCTORs

Example of a Database (with a Conceptual Data Model)

• **Some mini-world *relationships:***

SECTIONs *are of specific COURSEs*

STUDENTs *take SECTIONs*

COURSEs *have prerequisite COURSEs*

INSTRUCTORs *teach SECTIONs*

COURSEs *are offered by DEPARTMENTs*

STUDENTs *major in DEPARTMENTs*

Architecture of DBMS

A commonly used views of data approach is the three-level architecture suggested by ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee). ANSI/SPARC produced an interim report in 1972 followed by a final report in 1977. The reports proposed an architectural framework for databases. Under this approach, a database is considered as containing data about an *enterprise*. The three levels of the architecture are three different views of the data:

External - individual user view

Conceptual - community user view

Internal - physical or storage view

The three level database architecture allows a clear separation of the information meaning (conceptual view) from the external data representation and from the physical data structure layout. A database system that is able to separate the three different views of data is likely to be flexible and adaptable. This flexibility and adaptability is data independence that we have discussed earlier. We now briefly discuss the three different views.

The external level is the view that the individual user of the database has. This view is often a restricted view of the database and the same database may provide a number of different views for different classes of users. In general, the end users and even the application programmers are only interested in a subset of the database. For example, a department head may only be interested in the departmental finances and student enrolments but not the library information. The librarian would not be expected to have any interest in the information about academic staff. The payroll office would have no interest in student enrolments.

The conceptual view is the information model of the enterprise and contains the view of the whole enterprise without any concern for the physical implementation. This view is normally more stable than the other two views. In a database, it may be desirable to change the internal view to improve performance while there has been no change in the conceptual view of the

database. The conceptual view is the overall community view of the database and it includes all the information that is going to be represented in the database. The conceptual view is defined by the conceptual schema which includes definitions of each of the various types of data. The internal view is the view about the actual physical storage of data. It tells us what data is stored in the database and how. At least the following aspects are considered at this level:
Storage allocation e.g. B-trees, hashing etc.

Access paths e.g. specification of primary and secondary keys, indexes and pointers and sequencing. Miscellaneous e.g. data compression and encryption techniques, optimization of the internal structures. Efficiency considerations are the most important at this level and the data structures are chosen to provide an efficient database. The internal view does not deal with the physical devices directly. Instead it views a physical device as a collection of physical pages and allocates space in terms of logical pages.

The separation of the conceptual view from the internal view enables us to provide a logical description of the database without the need to specify physical structures. This is often called *physical data independence*. Separating the external views from the conceptual view enables us to change the conceptual view without affecting the external views. This separation is sometimes called *logical data independence*.

1.8 Data Independence

Data independence can be defined as the capacity to change the schema at one level without changing the schema at next higher level. There are two types of data Independence. They are
1. Logical data independence. 2. Physical data independence.
1. Logical data independence is the capacity to change the conceptual schema without having to change the external schema.
2. Physical data independence is the capacity to change the internal schema without changing the conceptual schema.

When not to use a DBMS

• Main inhibitors (costs) of using a DBMS:

• High initial investment and possible need for additional hardware.

• Overhead for providing generality, security, concurrency control, recovery, and integrity functions When a DBMS may be unnecessary:

• If the database and applications are simple, well defined and not expected to change.

• If there are stringent real-time requirements that may not be met because of DBMS overhead.

• If access to data by multiple users is not required. When no DBMS may suffice:

• If the database system is not able to handle the complexity of data because of modeling limitations.

• If the database users need special operations not supported by the DBMS.

Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of *data abstraction* by hiding details of data storage that are irrelevant to database users.

A **data model** ---a collection of concepts that can be used to describe the conceptual/logical structure of a database--- provides the necessary means to achieve this abstraction.

By *structure* is meant the data types, relationships, and constraints that should hold for the data.

Most data models also include a set of **basic operations** for specifying retrievals/updates.

Object-oriented data models include the idea of objects having behavior (i.e., applicable methods) being stored in the database (as opposed to purely "passive" data).

According to C.J. Date (one of the leading database experts), a **data model** is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users interact. The objects allow us to model the *structure* of data; the operators allow us to model its *behavior*.

In the *relational* data model, data is viewed as being organized in two-dimensional tables comprised of tuples of attribute values. This model has operations such as Project, Select, and Join.

A data model is not to be confused with its **implementation**, which is a physical realization on a real machine of the components of the *abstract machine* that together constitute that model.

Logical vs. physical!!

There are other well-known data models that have been the basis for database systems. The best-known models pre-dating the relational model are the **hierarchical** (in which the entity types form a tree) and the **network** (in which the entity types and relationships between them form a graph).

Categories of Data Models (based on degree of abstractness):

- **high-level/conceptual**: (e.g., ER model of Chapter 3) provides a view close to the way users would perceive data; uses concepts such as
    - **entity**: real-world object or concept (e.g., student, employee, course, department, event)
    - **attribute**: some property of interest describing an entity (e.g., height, age, color)
    - **relationship**: an interaction among entities (e.g., works-on relationship between an employee and a project)
- **representational/implementational**: intermediate level of abstractness; example is relational data model (or the network model alluded to earlier). Also called **record-based** model.
- **low-level/physical**: gives details as to how data is stored in computer system, such as record formats, orderings of records, access paths (indexes). (See Chapters 13-14.)

Schemas, Instances, and Database State

One must distinguish between the *description* of a database and the database itself. The former is called the **database schema**, which is specified during design and is not expected to change often. (See Figure 2.1, p. 33, for schema diagram for relational UNIVERSITY database.)

The actual data stored in the database probably changes often. The data in the database at a particular time is called the **state** of the database, or a **snapshot**.

Application requirements change occasionally, which is one of the reasons why software maintenance is important. On such occasions, a change to a database's schema may be called for. An example would be to add a Date_of_Birth field/attribute to the STUDENT table. Making changes to a database schema is known as **schema evolution**. Most modern DBMS's support schema evolution operations that can be applied while a database is operational.

DBMS Architecture and Data Independence

Three-Schema Architecture: This idea was first described by the ANSI/SPARC committee in late 1970's. The goal is to separate (i.e., insert layers of "insulation" between) user applications and the physical database. C.J. Date points out that it is an ideal that few, if any, real-life DBMS's achieve fully.

- **internal level**: has an internal/physical schema that describes the physical storage structure of the database using a low-level data model)
- **conceptual level**: has a conceptual schema describing the (logical) structure of the whole database for a community of users. It hides physical storage details, concentrating upon describing entities, data types, relationships, user operations, and constraints. Can be described using either high-level or implementational data model.
- **external/view level**: includes a number of external schemas (or user views), each of which describes part of the database that a particular category of users is interested in, hiding rest of database. Can be described using either high-level or implementational data model. (In practice, usually described using same model as is the conceptual schema.)

Users (including application programs) submit queries that are expressed with respect to the external level. It is the responsibility of the DBMS to **transform** such a query into one that is expressed with respect to the internal level (and to transform the result, which is at the internal level, into its equivalent at the external level).

Example: Select students with GPA > 1.15.

Q:How is this accomplished?

A: By virtue of **mappings** between the levels:

- **external/conceptual** mapping (providing **logical** data independence)
- **conceptual/internal** mapping (providing **physical** data independence)

Data independence is the capacity to change the schema at one level of the architecture without having to change the schema at the next higher level. We distinguish between **logical** and **physical** data independence according to which two adjacent levels are involved. The former refers to the ability to change the conceptual schema without changing the external schema. The latter refers to the ability to change the internal schema without having to change the conceptual.
For an **example of physical data independence**, suppose that the internal schema is modified (because we decide to add a new index, or change the encoding scheme used in representing some field's value, or stipulate that some previously unordered file must be ordered by a particular field). Then we can change the mapping between the conceptual and internal schemas in order to avoid changing the conceptual schema itself.
Not surprisingly, the process of transforming data via mappings can be costly (performance-wise), which is probably one reason that real-life DBMS's don't fully implement this 3-schema architecture.

Database Languages and Interfaces

A DBMS supports a variety of users and must provide appropriate languages and interfaces for
each category of users.

DBMS Languages

- DDL (Data Definition Language): used (by the DBA and/or database designers) to specify the conceptual schema.
- SDL (Storage Definition Language): used for specifying the internal schema
- VDL (View Definition Language): used for specifying the external schemas (i.e., user views)

- DML (Data Manipulation Language): used for performing operations such as retrieval and update upon the populated database

The above description represents some kind of ideal. In real-life, at least so far, the de facto standard DBMS language is SQL (Standard Query Language), which has constructs to support the functions needed by DDL, VDL, and DML languages. (Early versions of SQL had features in support of SDL functions, but no more.)

DBMS Interfaces

- Menu-based interfaces for web clients or browsing
- Forms-based interfaces
- GUI's
- Natural Language Interfaces
- Speech Input and Output
- Interfaces for parametric users
- Interfaces for the DBA

Classification of DBMS's

Based upon

- underlying data model (e.g., relational, object, object-relational, network)
- multi-user vs. single-user
- centralized vs. distributed
- cost
- general-purpose vs. special-purpose
- types of **access path** options

Outline of Database Design

The main phases of database design are depicted in Figure 3.1, page 59:

- **Requirements Collection and Analysis**: purpose is to produce a description of the users' requirements.

- **Conceptual Design**: purpose is to produce a *conceptual schema* for the database, including detailed descriptions of *entity types*, *relationship types*, and *constraints*. All these are expressed in terms provided by the data model being used. (*Remark*: As the ER model is focused on precisely these three concepts, it would seem that the authors are predisposed to using that data model!)
- **Implementation**: purpose is to transform the conceptual schema (which is at a high/abstract level) into a (lower-level) *representational/implementational* model supported by whatever DBMS is to be used.
- **Physical Design**: purpose is to decide upon the internal storage structures, access paths (indexes), etc., that will be used in realizing the representational model produced in previous phase.

Entity-Relationship (ER) Model

Our focus now is on the second phase, **conceptual design**, for which The **Entity-Relationship (ER) Model** is a popular high-level conceptual data model.

In the ER model, the main concepts are **entity**, **attribute**, and **relationship**.

Entities and Attributes

Entity: An entity represents some "thing" (in the miniworld) that is of interest to us, i.e., about which we want to maintain some data. An entity could represent a physical object (e.g., house, person, automobile, widget) or a less tangible concept (e.g., company, job, academic course).

Attribute: An entity is described by its attributes, which are properties characterizing it. Each attribute has a **value** drawn from some **domain** (set of meaningful values).

Example: A *PERSON* entity might be described by *Name*, *BirthDate*, *Sex*, etc., attributes, each having a particular value.

What distinguishes an entity from an attribute is that the latter is strictly for the purpose of describing the former and is not, in and of itself, of interest to us. It is sometimes said that an entity has an independent existence, whereas an attribute does not. In performing data modeling, however, it is not always clear whether a particular concept deserves to be classified as an entity or "only" as an attribute.

We can classify attributes along these dimensions:

- simple/atomic vs. composite
- single-valued vs. multi-valued (or set-valued)
- stored vs. derived (*Note from instructor:* this seems like an implementational detail that ought not be considered at this (high) level of abstraction.)

A **composite** attribute is one that is *composed* of smaller parts. An **atomic** attribute is indivisible or indecomposable.

- **Example 1**: A *BirthDate* attribute can be viewed as being composed of (sub-)attributes for month, day, and year.
- **Example 2**: An *Address* attribute (Figure 1.14, page 64) can be viewed as being composed of (sub-)attributes for street address, city, state, and zip code. A street address can itself be viewed as being composed of a number, street name, and apartment number. As this suggests, composition can extend to a depth of two (as here) or more.

To describe the structure of a composite attribute, one can draw a. In case we are limited to using text, it is customary to write its name followed by a parenthesized list of its sub-attributes. For the examples mentioned above, we would write

BirthDate(Month, Day, Year)
Address(StreetAddr(StrNum, StrName, AptNum), City, State, Zip)

Single- vs. multi-valued attribute: Consider a *PERSON* entity. The person it represents has (one) *SSN*, (one) *date of birth*, (one, although composite) *name*, etc. But that person may have zero or more academic degrees, dependents, or (if the person is a male living in Utah) spouses! How can we model this via attributes *AcademicDegrees*, *Dependents*, and *Spouses*? One way is to allow such attributes to be *multi-valued* (perhaps *set-valued* is a better term), which is to say that we assign to them a (possibly empty) *set* of values rather than a single value.

To distinguish a multi-valued attribute from a single-valued one, it is customary to enclose the former within curly braces (which makes sense, as such an attribute has a value that is a set, and

curly braces are traditionally used to denote sets). Using the *PERSON* example from above, we would depict its structure in text as

PERSON(SSN, Name, BirthDate(Month, Day, Year), { AcademicDegrees(School, Level, Year) }, { Dependents }, ...)

Here we have taken the liberty to assume that each academic degree is described by a school, level (e.g., B.S., Ph.D.), and year. Thus, *AcademicDegrees* is not only multi-valued but also composite. We refer to an attribute that involves some combination of multi-valuedness *and* compositeness as a **complex** attribute.

The structure of this attribute allows for the business to have several offices, each described by an address and a set of phone numbers that ring into that office. Its structure is given by
{ AddressPhone({ Phone(AreaCode, Number) }, Address(StrAddr(StrNum, StrName, AptNum), City, State, Zip)) }

Stored vs. derived attribute: Perhaps *independent* and *derivable* would be better terms for these (or *non-redundant* and *redundant*). In any case, a *derived* attribute is one whose value can be calculated from the values of other attributes, and hence need not be stored. **Example:** *Age* can be calculated from *BirthDate*, assuming that the current date is accessible.

The Null value: In some cases a particular entity might not have an applicable value for a particular attribute. Or that value may be unknown. Or, in the case of a multi-valued attribute, the appropriate value might be the empty set.
Example: The attribute *DateOfDeath* is not applicable to a living person and its correct value may be unknown for some persons who have died.
In such cases, we use a special attribute value (non-value?), called **null**. There has been some argument in the database literature about whether a different approach (such as having distinct values for *not applicable* and *unknown*) would be superior.

Entity Types, Entity Sets, Keys, and Domains

Above we mentioned the concept of a *PERSON* entity, i.e., a representation of a particular person via the use of attributes such as *Name*, *Sex*, etc. Chances are good that, in a database in which one such entity exists, we will want many others of the same kind to exist also, each of them described by the same collection of attributes. Of course, the *values* of those attributes will differ from one entity to another (e.g., one person will have the name "Mary" and another will have the name "Rumpelstiltskin"). Just as likely is that we will want our database to store information about other kinds of entities, such as business transactions or academic courses, which will be described by entirely different collections of attributes.

This illustrates the distinction between entity types and entity instances. An **entity type** serves as a template for a collection of **entity instances**, all of which are described by the same collection of attributes. That is, an entity type is analogous to a **class** in object-oriented programming and an entity instance is analogous to a particular object (i.e., instance of a class).
In ER modeling, we deal only with entity types, not with instances. In an ER diagram, each entity type is denoted by a rectangular box.

An **entity set** is the collection of all entities of a particular type that exist, in a database, at some moment in time.

Key Attributes of an Entity Type: A minimal collection of attributes (often only one) that, by design, distinguishes any two (simultaneously-existing) entities of that type. In other words, if attributes A_1 through A_m together form a key of entity type *E*, and *e* and *f* are two entities of type *E* existing at the same time, then, in at least one of the attributes A_i ($0 < i <= m$), *e* and *f* must have distinct values.

An entity type could have more than one key.

the CAR entity type is postulated to have both { *Registration(RegistrationNum, State)* } and { *VehicleID* } as keys.)

Domains (Value Sets) of Attributes: The domain of an attribute is the "universe of values" from which its value can be drawn. In other words, an attribute's domain specifies its set of allowable values. The concept is similar to **data type**.

Example Database Application: COMPANY

Suppose that Requirements Collection and Analysis results in the following (informal) description of the COMPANY miniworld:

The company is organized as a collection of **departments**.

- Each department
  - has a unique name
  - has a unique number
  - is associated with a set of locations
  - has a particular employee who acts as its manager (and who assumed that position on some date)
  - has a set of employees assigned to it
  - controls a set of projects
- Each project
  - has a unique name
  - has a unique number
  - has a single location
  - has a set of employees who work on it
  - is controlled by a single department
- Each employee
  - has a name
  - has a SSN that uniquely identifies her/him
  - has an address
  - has a salary
  - has a sex
  - has a birthdate

- has a direct supervisor
- has a set of dependents
- is assigned to one department
- works some number of hours per week on each of a set of projects (which need not all be controlled by the same department)

- Each dependent
  - has first name
  - has a sex
  - has a birthdate
  - is related to a particular employee in a particular way (e.g., child, spouse, pet)
  - is uniquely identified by the combination of her/his first name and the employee of which (s)he is a dependent

Initial Conceptual Design of COMPANY database

Using the above structured description as a guide, we get the following preliminary design for entity types and their attributes in the COMPANY database:

- DEPARTMENT(Name, Number, { Locations }, Manager, ManagerStartDate, { Employees }, { Projects })
- PROJECT(Name, Number, Location, { Workers }, ControllingDept)
- EMPLOYEE(Name(FName, MInit, LName), SSN, Sex, Address, Salary, BirthDate, Dept, Supervisor, { Dependents }, { WorksOn(Project, Hours) })
- DEPENDENT(Employee, FirstName, Sex, BirthDate, Relationship)

Remarks: Note that the attribute *WorksOn* of EMPLOYEE (which records on which projects the employee works) is not only multi-valued (because there may be several such projects) but also composite, because we want to record, for each such project, the number of hours per week that the employee works on it. Also, each *candidate key* has been indicated by underlining.

For similar reasons, the attributes *Manager* and *ManagerStartDate* of DEPARTMENT really ought to be combined into a single composite attribute. Not doing so causes little or no harm, however, because these are single-valued attributes. Multi-valued attributes would pose some

difficulties, on the other hand. Suppose, for example, that a department could have two or more managers, and that some department had managers Mary and Harry, whose start dates were 10-4-1999 and 1-13-2001, respectively. Then the values of the *Manager* and *ManagerStartDate* attributes should be *{ Mary, Harry }* and *{ 10-4-1999, 1-13-2001 }*. But from these two attribute values, there is no way to determine which manager started on which date. On the other hand, by recording this data as a set of ordered pairs, in which each pair identifies a manager and her/his starting date, this deficiency is eliminated. *End of Remarks*

Relationship Types, Sets, Roles, and Structural Constraints

Having presented a preliminary database schema for COMPANY, it is now convenient to clarify the concept of a **relationship** (which is the last of the three main concepts involved in the ER model).

Relationship: This is an association between two entities. As an example, one can imagine a STUDENT entity being associated to an ACADEMIC_COURSE entity via, say, an ENROLLED_IN relationship.

Whenever an attribute of one entity type refers to an entity (of the same or different entity type), we say that a relationship exists between the two entity types.

From our preliminary COMPANY schema, we identify the following **relationship types** (using descriptive names and ordering the participating entity types so that the resulting phrase will be in active voice rather than passive):

- EMPLOYEE MANAGES DEPARTMENT (arising from *Manager* attribute in DEPARTMENT)
- DEPARTMENT CONTROLS PROJECT (arising from *ControllingDept* attribute in PROJECT and the *Projects* attribute in DEPARTMENT)
- EMPLOYEE WORKS_FOR DEPARTMENT (arising from *Dept* attribute in EMPLOYEE and the *Employees* attribute in DEPARTMENT)
- EMPLOYEE SUPERVISES EMPLOYEE (arising from *Supervisor* attribute in EMPLOYEE)

- EMPLOYEE WORKS_ON PROJECT (arising from *WorksOn* attribute in EMPLOYEE and the *Workers* attribute in PROJECT)
- DEPENDENT DEPENDS_ON EMPLOYEE (arising from *Employee* attribute in DEPENDENT and the *Dependents* attribute in EMPLOYEE)

In ER diagrams, relationship types are drawn as diamond-shaped boxes connected by lines to the entity types involved. See Figure 3.2, page 62. Note that attributes are depicted by ovals connected by lines to the entity types they describe (with multi-valued attributes in double ovals and composite attributes depicted by trees). The original attributes that gave rise to the relationship types are absent, having been replaced by the relationship types.

A **relationship set** is a set of instances of a relationship type. If, say, R is a relationship type that relates entity types A and B, then, at any moment in time, the relationship set of R will be a set of ordered pairs *(x,y)*, where *x* is an instance of A and *y* is an instance of B. What this means is that, for example, if our COMPANY miniworld is, at some moment, such that employees e_1, e_3, and e_6 work for department d_1, employees e_2 and e_4 work for department d_2, and employees e_5 and e_7 work for department d_3, then the WORKS_FOR **relationship set** will include as **instances** the ordered pairs (e_1, d_1), (e_2, d_2), (e_3, d_1), (e_4, d_2), (e_5, d_3), (e_6, d_1), and (e_7, d_3).

Ordering of entity types in relationship types: Note that the order in which we list the entity types in describing a relationship is of little consequence, except that the relationship name (for purposes of clarity) ought to be consistent with it. For example, if we swap the two entity types in each of the first two relationships listed above, we should rename them IS_MANAGED_BY and IS_CONTROLLED_BY, respectively.

Degree of a relationship type: Also note that, in our COMPANY example, all relationship instances will be ordered pairs, as each relationship associates an instance from one entity type with an instance of another (or the same, in the case of SUPERVISES) relationship type. Such relationships are said to be *binary*, or to have *degree* two. Relationships with degree three (called *ternary*) or more are also possible, although not as common. This is illustrated in Figure 3.10 (page 72), where a relationship SUPPLY (perhaps not the best choice for a name) has as

instances ordered triples of suppliers, parts, and projects, with the intent being that inclusion of the ordered triple (s_2, p_4, j_1), for example, indicates that supplier s_2 supplied part p_4 to project j_1).

Roles in relationships: Each entity that participates in a relationship plays a particular *role* in that relationship, and it is often convenient to refer to that role using an appropriate name. For example, in each instance of a WORKS_FOR relationship set, the employee entity plays the role of *worker* or (surprise!) *employee* and each department plays the role of *employer* or (surprise!) *department*. Indeed, as this example suggests, often it is best to use the same name for the role as for the corresponding entity type.

An exception to this rule occurs when the same entity type plays two (or more) roles in the same relationship. (Such relationships are said to be *reCURsive*, which I find to be a misleading use of that term. A better term might be *self-referential*.) For example, in each instance of a SUPERVISES relationship set, one employee plays the role of *supervisor* and the other plays the role of *supervisee*.

Constraints on Relationship Types

Often, in order to make a relationship type be an accurate model of the miniworld concepts that it is intended to represent, we impose certain constraints that limit the possible corresponding relationship sets. (That is, a constraint may make "invalid" a particular set of instances for a relationship type.)

There are two main kinds of relationship constraints (on binary relationships). For illustration, let *R* be a relationship set consisting of ordered pairs of instances of entity types *A* and *B*, respectively.

- **cardinality ratio**:
    - **1:1 (one-to-one)**: Under this constraint, no instance of *A* may particpate in more than one instance of *R*; similarly for instances of *B*. In other words, if (a_1, b_1) and (a_2, b_2) are (distinct) instances of *R*, then neither $a_1 = a_2$ nor $b_1 = b_2$. **Example**: Our informal description of COMPANY says that every department has one employee who manages it. If we also stipulate that an employee may not

(simultaneously) play the role of manager for more than one department, it follows that MANAGES is 1:1.

- **1:N (one-to-many)**: Under this constraint, no instance of *B* may participate in more than one instance of *R*, but instances of *A* are under no such restriction. In other words, if (a_1, b_1) and (a_2, b_2) are (distinct) instances of *R*, then it cannot be the case that $b_1 = b_2$.
  Example: CONTROLS is 1:N because no project may be controlled by more than one department. On the other hand, a department may control any number of projects, so there is no restriction on the number of relationship instances in which a particular department instance may participate. For similar reasons, SUPERVISES is also 1:N.
- **N:1 (many-to-one)**: This is just the same as 1:N but with roles of the two entity types reversed.
  Example: WORKS_FOR and DEPENDS_ON are N:1.
- **M:N (many-to-many)**: Under this constraint, there are no restrictions. (Hence, the term applies to the absence of a constraint!)
  Example: WORKS_ON is M:N, because an employee may work on any number of projects and a project may have any number of employees who work on it.

Notice the notation in Figure 3.2 for indicating each relationship type's cardinality ratio.

Suppose that, in designing a database, we decide to include a binary relationship *R* as described above (which relates entity types *A* and *B*, respectively). To determine how *R* should be constrained, with respect to cardinality ratio, the questions you should ask are these:

May a given entity of type B be related to multiple entities of type A?
May a given entity of type A be related to multiple entities of type B?

The pair of answers you get maps into the four possible cardinality ratios as follows:

(yes, yes) --> M:N
(yes, no) --> N:1
(no, yes) --> 1:N
(no, no) --> 1:1

- **participation**: specifies whether or not the existence of an entity depends upon its being related to another entity via the relationship.
    - **total participation (or existence dependency)**: To say that entity type *A* is constrained to **participate totally** in relationship *R* is to say that if (at some moment in time) *R*'s instance set is

      { (a_1, b_1), (a_2, b_2), ... (a_m, b_m) },

      then (at that same moment) *A*'s instance set must be *{ a_1, a_2, ..., a_m }*. In other words, there can be no member of *A*'s instance set that does not participate in at least one instance of *R*.

      According to our informal description of COMPANY, every employee must be assigned to some department. That is, every employee instance must participate in at least one instance of WORKS_FOR, which is to say that *EMPLOYEE* satisfies the total participation constraint with respect to the WORKS_FOR relationship.

      In an ER diagram, if entity type *A* must participate totally in relationship type R, the two are connected by a double line. See Figure 3.2.
    - **partial participation**: the absence of the total participation constraint! (E.g., not every employee has to participate in MANAGES; hence we say that, with respect to MANAGES, *EMPLOYEE* participates partially. This is not to say that for all employees to be managers is not allowed; it only says that it need not be the case that all employees are managers.

Attributes of Relationship Types

Relationship types, like entity types, can have attributes. A good example is WORKS_ON, each instance of which identifies an employee and a project on which (s)he works. In order to record (as the specifications indicate) how many hours are worked by each employee on each project, we include *Hours* as an attribute of WORKS_ON. (See Figure 3.2 again.) In the case of an M:N relationship type (such as WORKS_ON), allowing attributes is vital. In the case of an N:1, 1:N, or 1:1 relationship type, any attributes can be assigned to the

entity type opposite from the 1 side. For example, the *StartDate* attribute of the MANAGES relationship type can be given to either the *EMPLOYEE* or the *DEPARTMENT* entity type.

Weak Entity Types: An entity type that has no set of attributes that qualify as a key is called **weak**. (Ones that do are **strong**.)
An entity of a weak identity type is uniquely identified by the specific entity to which it is related (by a so-called **identifying relationship** that relates the weak entity type with its so-called **identifying** or **owner entity type**) in combination with some set of its own attributes (called a *partial key*).

Example: A *DEPENDENT* entity is identified by its first name together with the *EMPLOYEE* entity to which it is related via DEPENDS_ON. (Note that this wouldn't work for former heavyweight boxing champion George Foreman's sons, as they all have the name "George"!)
Because an entity of a weak entity type cannot be identified otherwise, that type has a **total participation constraint** (i.e., **existence dependency**) with respect to the identifying relationship.

This should not be taken to mean that any entity type on which a total participation constraint exists is weak. For example, DEPARTMENT has a total participation constraint with respect to MANAGES, but it is not weak.
In an ER diagram, a weak entity type is depicted with a double rectangle and an identifying relationship type is depicted with a double diamond.

Design Choices for ER Conceptual Design:

Sometimes it is not clear whether a particular miniworld concept ought to be modeled as an entity type, an attribute, or a relationship type. Here are some guidelines (given with the understanding that schema design is an iterative process in which an initial design is refined repeatedly until a satisfactory result is achieved):

- As happened in our development of the ER model for COMPANY, if an attribute of entity type *A* serves as a reference to an entity of type *B*, it may be wise to refine that attribute into a binary relationship involving entity types *A* and *B*. It may well be that *B* has a corresponding attribute referring back to *A*, in which case it, too, is refined into the aforementioned relationship. In our COMPANY example, this was exemplified by the *Projects* and *ControllingDept* attributes of *DEPARTMENT* and *PROJECT*, respectively.
- An attribute that exists in several entity types may be refined into its own entity type. For example, suppose that in a UNIVERSITY database we have entity types *STUDENT*, *INSTRUCTOR*, and *COURSE*, all of which have a *Department* attribute. Then it may be wise to introduce a new entity type, *DEPARTMENT*, and then to follow the preceding guideline by introducing a binary relationship between *DEPARTMENT* and each of the three aforementioned entity types.
- An entity type that is involved in very few relationships (say, zero, one, or possibly two) could be refined into an attribute (of each entity type to which it is related).

Module 2

The Relational Data Model and Relational Database Constraints and Relational Algebra Origins.

Relational Model Concepts

- **Domain**: A (usually named) set/universe of *atomic* values, where by "atomic" we mean simply that, from the point of view of the database, each value in the domain is indivisible (i.e., cannot be broken down into component parts).
  Examples of domains (some taken from page 147):
  - USA_phone_number: string of digits of length ten
  - SSN: string of digits of length nine
  - Name: string of characters beginning with an upper case letter
  - GPA: a real number between 0.0 and 4.0
  - Sex: a member of the set { female, male }
  - Dept_Code: a member of the set { CMPS, MATH, ENGL, PHYS, PSYC, ... }

These are all *logical* descriptions of domains. For implementation purposes, it is necessary to provide descriptions of domains in terms of concrete **data types** (or **formats**) that are provided by the DBMS (such as String, int, boolean), in a manner analogous to how programming languages have intrinsic data types.

- **Attribute**: the *name* of the role played by some value (coming from some domain) in the context of a **relational schema**. The domain of attribute A is denoted dom(A).
- **Tuple**: A tuple is a mapping from attributes to values drawn from the respective domains of those attributes. A tuple is intended to describe some entity (or relationship between entities) in the miniworld.
  As an example, a tuple for a PERSON entity might be
  { Name --> "Rumpelstiltskin", Sex --> Male, IQ --> 143 }

- **Relation**: A (named) set of tuples all of the same form (i.e., having the same set of attributes). The term **table** is a loose synonym. (Some database purists would argue that a table is "only" a physical manifestation of a relation.)
- **Relational Schema**: used for describing (the structure of) a relation. E.g., R(A_1, A_2, ..., A_n) says that R is a relation with *attributes* A_1, ... A_n. The **degree** of a relation is the number of attributes it has, here *n*.

  Example: STUDENT(Name, SSN, Address)

  (See Figure 2.1, page 149, for an example of a STUDENT relation/table having several tuples/rows.)

  One would think that a "complete" relational schema would also specify the domain of each attribute.
- **Relational Database**: A collection of **relations**, each one consistent with its specified relational schema.

Characteristics of Relations

Ordering of Tuples: A relation is a *set* of tuples; hence, there is no order associated with them. That is, it makes no sense to refer to, for example, the 5th tuple in a relation. When a relation is depicted as a table, the tuples are necessarily listed in *some* order, of course, but you should attach no significance to that order. Similarly, when tuples are represented on a storage device, they must be organized in *some* fashion, and it may be advantageous, from a performance standpoint, to organize them in a way that depends upon their content.

Ordering of Attributes: A tuple is best viewed as a mapping from its attributes (i.e., the names we give to the roles played by the values comprising the tuple) to the corresponding values. Hence, the order in which the attributes are listed in a table is irrelevant. (Note that, unfortunately, the set theoretic operations in relational algebra (at least how E&N define them) make implicit use of the order of the attributes. Hence, E&N view attributes as being arranged as a sequence rather than a set.)

Values of Attributes: For a relation to be in *First Normal Form*, each of its attribute domains must consist of atomic (neither composite nor multi-valued) values. Much of the theory underlying the relational model was based upon this assumption. Chapter 10 addresses the issue of including non-atomic values in domains. (Note that in the latest edition of C.J. Date's book, he explicitly argues against this idea, admitting that he has been mistaken in the past.)
The **Null** value: used for *don't know, not applicable.*

Interpretation of a Relation: Each relation can be viewed as a **predicate** and each tuple in that relation can be viewed as an assertion for which that predicate is satisfied (i.e., has value **true**) for the combination of values in it. In other words, each tuple represents a fact. Example (see Figure 2.1): The first tuple listed means: There exists a student having name Benjamin Bayer, having SSN 305-61-2435, having age 19, etc.
Keep in mind that some relations represent facts about entities (e.g., students) whereas others represent facts about relationships (between entities). (e.g., students and course sections).
The **closed world assumption** states that the only true facts about the miniworld are those represented by whatever tuples currently populate the database.

Relational Model Notation:

- $R(A_1, A_2, ..., A_n)$ is a relational schema of degree n denoting that there is a relation R having as its attributes $A_1, A_2, ..., A_n$.
- By convention, Q, R, and S denote relation names.
- By convention, q, r, and s denote relation states. For example, $r(R)$ denotes one possible state of relation R. If R is understood from context, this could be written, more simply, as r.
- By convention, t, u, and v denote tuples.
- The "dot notation" $R.A$ (e.g., STUDENT.Name) is used to qualify an attribute name, usually for the purpose of distinguishing it from a same-named attribute in a different relation (e.g., DEPARTMENT.Name).

Relational Model Constraints and Relational Database Schemas

Constraints on databases can be categorized as follows:

- **inherent model-based:** Example: no two tuples in a relation can be duplicates (because a relation is a set of tuples)
- **schema-based:** can be expressed using DDL; this kind is the focus of this section.
- **application-based:** are specific to the "business rules" of the miniworld and typically difficult or impossible to express and enforce within the data model. Hence, it is left to application programs to enforce.

Elaborating upon **schema-based constraints**:

Domain Constraints:

Each attribute value must be either **null** (which is really a *non-value*) or drawn from the domain of that attribute. Note that some DBMS's allow you to impose the **not null** constraint upon an attribute, which is to say that that attribute may not have the (non-)value **null**.

Key Constraints:

A relation is a *set* of tuples, and each tuple's "identity" is given by the values of its attributes. Hence, it makes no sense for two tuples in a relation to be identical (because then the two tuples are actually one and the same tuple). That is, no two tuples may have the same combination of values in their attributes.

Usually the miniworld dictates that there be (proper) subsets of attributes for which no two tuples may have the same combination of values. Such a set of attributes is called a **superkey** of its relation. From the fact that no two tuples can be identical, it follows that the set of all attributes of a relation constitutes a superkey of that relation.

A **key** is a *minimal superkey*, i.e., a superkey such that, if we were to remove any of its attributes, the resulting set of attributes fails to be a superkey.

Example: Suppose that we stipulate that a faculty member is uniquely identified by *Name* and *Address* and also by *Name* and *Department*, but by no single one of the three attributes mentioned. Then *{ Name, Address, Department }* is a (non-minimal) superkey and each of *{ Name, Address }* and *{ Name, Department }* is a key (i.e., minimal superkey).

Candidate key: any key! (Hence, it is not clear what distinguishes a key from a candidate key.)
Primary key: a key chosen to act as the means by which to identify tuples in a relation. Typically, one prefers a primary key to be one having as few attributes as possible.

Relational Databases and Relational Database Schemas

A **relational database schema** is a set of schemas for its relations together with a set of **integrity constraints**.
A **relational database state/instance/snapshot** is a set of states of its relations such that no integrity constraint is violated.

Entity Integrity, Referential Integrity, and Foreign Keys

Entity Integrity Constraint:

In a tuple, none of the values of the attributes forming the relation's primary key may have the (non-)value **null**. Or is it that at least one such attribute must have a non-null value? In my opinion, E&N do not make it clear!

Referential Integrity Constraint:

A **foreign key** of relation *R* is a set of its attributes intended to be used (by each tuple in *R*) for identifying/referring to a tuple in some relation *S*. (*R* is called the *referencing* relation and *S* the *referenced* relation.) For this to make sense, the set of attributes of *R* forming the foreign key should "correspond to" some superkey of *S*. Indeed, by definition we require this superkey to be the primary key of *S*.

This constraint says that, for every tuple in *R*, the tuple in *S* to which it refers must actually be in *S*. Note that a foreign key may refer to a tuple in the same relation and that a foreign key may be part of a primary key (indeed, for weak entity types, this will always occur). A foreign key may have value **null** (necessarily in all its attributes??), in which case it does not refer to any tuple in the referenced relation.

Semantic Integrity Constraints:

application-specific restrictions that are unlikely to be expressible in DDL. Examples:

- salary of a supervisee cannot be greater than that of her/his supervisor
- salary of an employee cannot be lowered

Update Operations and Dealing with Constraint Violations

For each of the *update* operations (Insert, Delete, and Update), we consider what kinds of constraint violations may result from applying it and how we might choose to react.

Insert:

- domain constraint violation: some attribute value is not of correct domain
- entity integrity violation: key of new tuple is **null**
- key constraint violation: key of new tuple is same as existing one
- referential integrity violation: foreign key of new tuple refers to non-existent tuple

Ways of dealing with it: reject the attempt to insert! Or give user opportunity to try again with different attribute values.

Delete:

- referential integrity violation: a tuple referring to the deleted one exists.

Three options for dealing with it:

- Reject the deletion
- Attempt to **cascade** (or **propagate**) by deleting any referencing tuples (plus those that reference them, etc., etc.)

- modify the foreign key attribute values in referencing tuples to **null** or to some valid value referencing a different tuple

Update:

- Key constraint violation: primary key is changed so as to become same as another tuple's
- referential integrity violation:
  - foreign key is changed and new one refers to nonexistent tuple
  - primary key is changed and now other tuples that had referred to this one violate the constraint

Transactions:

This concept is relevant in the context where multiple users and/or application programs are accessing and updating the database concurrently. A transaction is a logical unit of work that may involve several accesses and/or updates to the database (such as what might be required to reserve several seats on an airplane flight). The point is that, even though several transactions might be processed concurrently, the end result must be as though the transactions were carried out sequentially. (Example of simultaneous withdrawals from same checking account.)

Relational Model Concepts

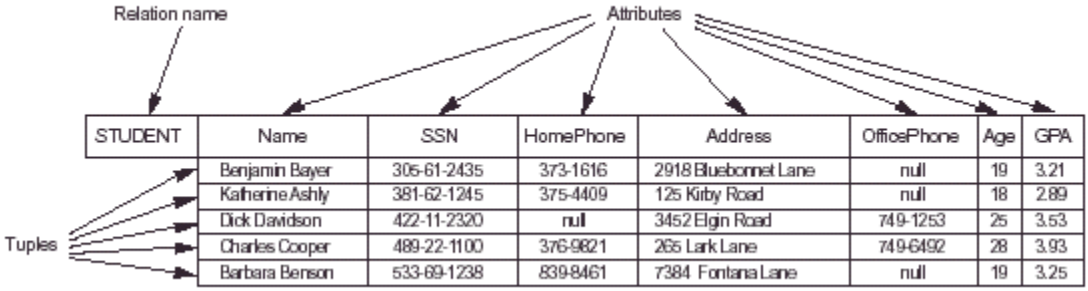
- The model was first introduced by Tod Codd of IBM Research in 1970.
- It uses the concept of a mathematical relation. Hence, the database is a collection of relations.
- A relation can be thought as a table of values, each row in the table represents a collection of related data values.
- In relational model terminology, a row is called a *tuple*, a column header is called an *attribute*.
- A **relation schema** *R*, denoted by $R(A_1,...,A_n)$, is made up of a relation name *R* and a list of attributes $A_1,...,A_n$.
- The domain of A_i is denoted by $dom(A_i)$.
- A relation schema that describes a relation *R* is called the *name* of this relation.

- The **degree** of a relation is the number of attributes in its relation schema.

For example, a relation schema of order 7

STUDENT(Name, SSN, HPhone, Address, WPhone, Age, GPA) describes students. The relation student can be shown as follows:

Fig: The attributes and tuples of a relation STUDENT



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

Characteristics of Relation

- A *tuple* can be considered as a set of (<attribute>, <value>) pairs. Thus the following two tuples are *identical*:
- t1 = <(Name, B. Bayer),(SSN, 305-61-2435),(HPhone, 3731616),
- (Address, 291 Blue Lane),(WPhone, null),(Age, 23),(GPA, 3.25)>
-
- t2 = <(HPhone, 3731616),(WPhone, null),(Name, B. Bayer),(Age, 23),
  (Address, 291 Blue Lane),(SSN, 305-61-2435),(GPA, 3.25)>
- *Tuple* ordering is not a part of relation, that is the following relation is *identical* to that of Table 7.1.

The relation STUDENT with a different order of tuples

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

Relational Model Notation

- A relation schema *R* of degree *n* is denoted by $R(A_1,...,A_n)$.

- An *n*-tuple *t* in a relation r(R) is dented by $t = < v_1,...,v_n >$, where
  - v_i is the value corresponding to attribute A_i .
  - Both $t[A_i]$ and $t.A_i$ refer to the value A_i .
- The letters *Q, R, S* denote relation names.
- The letters *q, r, s* denote relation states.
- The letters *t, u, v* denote tuples.
- An attribute *A* can be qualified with the relation name *R* using the dot notation *R.A* -for example, STUDENT.Name or STUDENT.Age.

The Relational Algebra

Operations to manipulate relations.

Used to specify retrieval requests (queries).

Query result is in the form of a relation

Relational Operations:

SELECT σ and PROJECT π operations.

Set operations: These include UNION U, INTERSECTION | |, DIFFERENCE -, CARTESIAN PRODUCT X.

JOIN operations $\bowtie$.

Other relational operations: DIVISION, OUTER JOIN, AGGREGATE FUNCTIONS.

SELECT σ and PROJECT π

SELECT operation (denoted by σ):

Selects the tuples (rows) from a relation R that satisfy a certain *selection condition* c

Form of the operation: σ_c .The condition c is an arbitrary Boolean expression on the attributes of R . Resulting relation has the *same attributes* as R. Resulting relation includes each tuple in r(R) whose attribute values satisfy the condition c

Examples:

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

$\sigma_{SALARY>30000}(EMPLOYEE)$

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

PROJECT operation (denoted by π):

Keeps only certain attributes (columns) from a relation R specified in an *attribute list* L .Form of operation: $\pi_L(R)$. Resulting relation has only those attributes of R specified in L .The PROJECT operation eliminates duplicate tuples in the resulting relation so that it remains a mathematical set (no duplicate elements). Duplicate tuples are eliminated by the π operation.

Example: $\pi_{SEX,SALARY}(EMPLOYEE)$

If several male employees have salary 30000, only a single tuple <M, 30000> is kept in the resulting relation.

Figure 7.8 Results of SELECT and PROJECT operations.
(a) $\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(EMPLOYEE)$.
(b) $\pi_{LNAME, FNAME, SALARY}(EMPLOYEE)$. (c) $\pi_{SEX, SALARY}(EMPLOYEE)$

(a)

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

(b)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

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

Sequences of operations:

Several operations can be combined to form a *relational algebra expression* (query)

Example: Retrieve the names and salaries of employees who work in department 4:

$$\pi_{FNAME,LNAME,SALARY}\ (\ \sigma_{DNO=4(EMPLOYEE)}\ )$$

Alternatively, we specify explicit intermediate relations for each step:

$$DEPT4\_EMPS \leftarrow \sigma_{DNO=4}(EMPLOYEE)$$

$$\rho \leftarrow \pi_{FNAME,\ LNAME,\ SALARY}(DEPT4\_EMPS)$$

Attributes can optionally be *renamed* in the resulting left-hand-side relation (this may be required for some operations that will be presented later):

$$DEPT4\_EMPS \leftarrow \sigma_{DNO=4}(EMPLOYEE)$$

$$\rho_{(FIRSTNAME,LASTNAME,SALARY)} \leftarrow \pi_{NAME,LNAME,SALARY}(DEPT4\_EMPS)$$

Figure 7.9 Results of relational algebra expressions. (a) $\pi_{LNAME,\ FNAME,\ SALARY}\ (\sigma_{DNO=5}(EMPLOYEE))$. (b) The same expression using intermediate relations and renaming of attributes.

(a)

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

(b)

TEMP	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren,Houston,TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
	Ramesh	K	Narayan	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

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

Set Operations

Binary operations from mathematical set theory:

UNION: $R_1 \cup R_2$, **INTERSECTION**: $R_1 \cap R_2$, **SET DIFFERENCE**: $R_1 - R_2$, **CARTESIAN PRODUCT**: $R_1 \times R_2$.

For ∪, ∩, -, the operand relations R1(A1, A2, ..., An) and R2(B1, B2, ..., Bn) must have the same number of attributes, and the domains of corresponding attributes must be compatible; that is, dom(Ai) = dom(Bi) for i=1, 2, ..., n. This condition is called union compatibility. The resulting relation for ∪, ∩, or - has the same attribute names as the first operand relation R1 (by convention).

Figure 7.11 Illustrating the set operations union, intersection, and difference. (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

CARTESIAN PRODUCT

$R(A_1, A_2, ..., A_m, B_1, B_2, ..., B_n) \leftarrow R_1(A_1, A_2, ..., A_m)$ X $R_2 (B_1, B_2, ..., B_n)$

A tuple t exists in R for each combination of tuples t_1 from R_1 and t_2 from R_2 such that:

$t[A_1, A_2, ..., A_m] = t_1$ and $t[B_1, B_2, ..., B_n] = t_2$

If R_1 has n_1 tuples and R_2 has n_2 tuples, then R will have $n_1 * n_2$ tuples.

CARTESIAN PRODUCT is a *meaningless operation* on its own. It can *combine related tuples* from two relations *if followed by the appropriate SELECT operation*.

Example: Combine each DEPARTMENT tuple with the EMPLOYEE tuple of the manager.

DEP_EMP $\leftarrow$ DEPARTMENT X EMPLOYEE

DEPT_MANAGER $\leftarrow \sigma_{MGRSSN=SSN}$(DEP_EMP)

Figure 7.12 An illustration of the CARTESIAN PRODUCT operation.

FEMALE_EMPS

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Alicia	J.	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S.	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A.	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

FNAME	LNAME	SSN
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05
Joyce	English	453453453	333445555	Alice	F	1986-04-05
Joyce	English	453453453	333445555	Theodore	M	1983-10-25
Joyce	English	453453453	333445555	Joy	F	1958-05-03
Joyce	English	453453453	987654321	Abner	M	1942-02-28
Joyce	English	453453453	123456789	Michael	M	1988-01-04
Joyce	English	453453453	123456789	Alice	F	1988-12-30
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05

ACTUAL_DEPENDENTS

FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28

RESULT

FNAME	LNAME	DEPENDENT_NAME
Jennifer	Wallace	Abner

JOIN Operations

THETA JOIN: Similar to a CARTESIAN PRODUCT followed by a SELECT. The condition c is called a *join condition*.

$R(A_1, A_2, ..., A_m, B_1, B_2, ..., B_n) \leftarrow R_1(A_1, A_2, ..., A_m) \bowtie_c R_2 (B_1, B_2, ..., B_n)$

EQUIJOIN: The join condition c includes one or more *equality comparisons* involving attributes from R_1 and R_2. That is, c is of the form:

$(A_i=B_j)$ AND ... AND $(A_h=B_k)$; $1 \leq i,h \leq m$, $1 \leq j,k \leq n$

In the above EQUIJOIN operation:

$A_i, ..., A_h$ are called the **join attributes** of R_1

$B_j, ..., B_k$ are called the **join attributes** of R_2

Example of using EQUIJOIN:

Retrieve each DEPARTMENT's name and its manager's name:

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

$RESULT \leftarrow \pi_{DNAME,FNAME,LNAME} (T)$

NATURAL JOIN (*):

In an EQUIJOIN $R \leftarrow R_1 \bowtie_C R_2$, the join attribute of R_2 appear *redundantly* in the result relation R. In a NATURAL JOIN, the *redundant join attributes* of R_2 are *eliminated* from R. The equality condition is *implied* and need not be specified.

$R \leftarrow R_1 *_{(\text{join attributes of R1}),(\text{join attributes of R2})} R_2$

Example: Retrieve each EMPLOYEE's name and the name of the DEPARTMENT he/she works for:

$T \leftarrow \text{EMPLOYEE} *_{(DNO),(DNUMBER)} \text{DEPARTMENT}$

$\text{RESULT} \leftarrow \pi_{FNAME,LNAME,DNAME}(T)$

If the join attributes *have the same names* in both relations, they *need not be specified* and we can write $R \leftarrow R_1 * R_2$.

Example: Retrieve each EMPLOYEE's name and the name of his/her SUPERVISOR:

$\text{SUPERVISOR(SUPERSSN,SFN,SLN)} \leftarrow \pi_{SSN,FNAME,LNAME}(\text{EMPLOYEE})$

$T \leftarrow \text{EMPLOYEE} * \text{SUPERVISOR}$

$\text{RESULT} \leftarrow \pi_{FNAME,LNAME,SFN,SLN}(T)$

Figure 7.14 An illustration of the NATURAL JOIN operation. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

(a)

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

(b)

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

Note: In the *original definition* of NATURAL JOIN, the join attributes were *required* to have the same names in both relations.

There can be a *more than one set of join attributes* with a *different meaning* between the same two relations. For example:

JOIN ATTRIBUTES

RELATIONSHIP

EMPLOYEE.SSN= EMPLOYEE *manage* DEPARTMENT.MGRSSN the DEPARTMENT

EMPLOYEE.DNO= EMPLOYEE *works for* DEPARTMENT.DNUMBER the DEPARTMENT

Example: Retrieve each EMPLOYEE's name and the name of the DEPARTMENT he/she works for:

$T \leftarrow EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT$

$RESULT \leftarrow \pi_{FNAME,LNAME,DNAME}(T)$

A relation can have a *set of join attributes* to join it with *itself* :

JOIN ATTRIBUTES RELATIONSHIP

EMPLOYEE(1).SUPERSSN= EMPLOYEE(2) *supervises*
EMPLOYEE(2).SSN EMPLOYEE(1)

One can *think of this* as joining *two distinct copies* of the relation, although only one relation actually exists In this case, *renaming* can be useful.

Example: Retrieve each EMPLOYEE's name and the name of his/her SUPERVISOR:

SUPERVISOR(SSSN,SFN,SLN) $\leftarrow \pi_{SSN,FNAME,LNAME}$(EMPLOYEE)

T$\leftarrow$EMPLOYEE $\bowtie_{SUPERSSN=SSSN}$SUPERVISOR

RESULT $\leftarrow \pi_{FNAME,LNAME,SFN,SLN}$(T)

Complete Set of Relational Algebra Operations:

All the operations discussed so far can be described as a sequence of *only* the operations SELECT, PROJECT, UNION, SET DIFFERENCE, and CARTESIAN PRODUCT.

Hence, the set {σ ,π , , - , X } is called a *complete set* of relational algebra operations. Any query language *equivalent to* these operations is called **relationally complete**.

For database applications, additional operations are needed that were not part of the *original* relational algebra. These include:

1. Aggregate functions and grouping.
2. OUTER JOIN and OUTER UNION.

Additional Relational Operations

AGGREGATE FUNCTIONS ($\Im$)

Functions such as SUM, COUNT, AVERAGE, MIN, MAX are often applied to sets of values or sets of tuples in database applications

<grouping attributes> $\Im_{<function\ list>}$(R)

The grouping attributes are optional

Example 1: Retrieve the average salary of all employees (no grouping needed):

ρ(AVGSAL)$\leftarrow \Im_{AVERAGE\ SALARY}$ (EMPLOYEE)

Example 2: For each department, retrieve the department number, the number of employees, and the average salary (in the department):

$\rho_{(DNO,NUMEMPS,AVGSAL)} \leftarrow$ DNO $\Im_{COUNT\ SSN,\ AVERAGE\ SALARY}$ (EMPLOYEE)

DNO is called the *grouping attribute* in the above example

Figure 7.16 An illustration of the AGGREGATE FUNCTION operation. (a) $R(DNO, NO\_OF\_EMPLOYEES, AVERAGE\_SAL) \leftarrow {}_{DNO}\Im_{COUNT\ SSN,AVERAGE\ SALARY}(EMPLOYEE)$. (b) ${}_{DNO}\Im_{COUNT\ SSN,AVERAGE\ SALARY}(EMPLOYEE)$. (c) $\Im_{COUNT\ SSN,AVERAGE\ SALARY}(EMPLOYEE)$.

(a)

	DNO	NO_OF_EMPLOYEES	AVERAGE_SAL
	5	4	33250
	4	3	31000
	1	1	55000

OUTER JOIN

In a regular EQUIJOIN or NATURAL JOIN operation, tuples in R_1 or R_2 that do not have matching tuples in the other relation *do not appear in the result*

Some queries require all tuples in R_1 (or R_2 or both) to appear in the result

When no matching tuples are found, **null**s are placed for the missing attributes

LEFT OUTER JOIN: R_1 X R_2 lets every tuple in R_1 appear in the result

RIGHT OUTER JOIN: R_1 X R_2 lets every tuple in R_2 appear in the result

FULL OUTER JOIN: R_1 X R_2 lets every tuple in R_1 or R_2 appear in the result

Figure 7.18 The LEFT OUTER JOIN operation.

RESULT	FNAME	MINIT	LNAME	DNAME
	John	B	Smith	null
	Franklin	T	Wong	Research
	Alicia	J	Zelaya	null
	Jennifer	S	Wallace	Administration
	Ramesh	K	Narayan	null
	Joyce	A	English	null
	Ahmad	V	Jabbar	null
	James	E	Borg	Headquarters

Data Definition, Constraints, and Schema Changes in SQL2

- *Structured Query Language (SQL)* was designed and implemented at IBM Research.
- Created in late 70's, under the name of SEQUEL
- A standard version of SQL (ANSI 1986), is called SQL86 or SQL1.
- A revised version of standard SQL, called SQL2 (or SQL92).
- SQL are going to be extended with objectoriented and other recent database concepts.
- Consists of
    - A Data Definition Language (DDL) for declaring database schemas
    - Data Manipulation Language (DML) for modifying and querying database instances
- In SQL, relation, tuple, and attribute are called *table*, *row*, and *columns* respectively.
- The SQL commands for data definition are *CREATE*, *ALTER*, and *DROP*.
- The *CREATE TABLE* Command is used to specify a new table by giving it a name and specifying its attributes (columns) and constraints.
- Data types available for attributes are:
    - *Numeric* integer, real (formated, such as *DECIMAL(10,2)*)
    - *CharacterString* fixedlength and varyinglength
    - *BitString* fixedlength, varyinglength
    - *Date* in the form YYYYMMDD
    - *Time* in the form HH:MM:SS
    - *Timestamp* includes both the *DATE* and *TIME* fields
    - *Interval* to increase/decrease the value of date, time, or timestamp

Basic Queries in SQL

- SQL allows a table (relation) to have two or more tuples that are identical in all their attributes values. Hence, an **SQL table** is not a set of tuple, because a set does not allow two identical members; rather it is a multiset of tuples.
- A basic query statement in SQL is the *SELECT* statement.
- The *SELECT* statement used in SQL has no relationship to the *SELECT* operation of relational algebra.

The SELECT Statement

The syntax of this command is:

SELECT <attribute list>

FROM <table list>

WHERE <Condition>;

Some example:

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

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

Query 1: Retrieve the name and address of all employee who work for the 'Research' Dept.

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

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

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

Dealing with Ambiguous Attribute Names and Renaming (Aliasing)

Ambiguity in the case where attributes are same name need to qualify the attribute using DOT separator

e.g., WHERE DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER

More Ambiguity in the case of queries that refer to the same relation twice

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

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

Unspecified WHERE Clause and Use of Asterisk (*)

A missing WHERE clause indicates no conditions, which means all tuples are selected

In case of two or more table, then all possible tuple combinations are selected

Example: Q10: Select all EMPLOYEE SSNs , and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME

```
SELECT   SSN, DNAME
FROM      EMPLOYEE, DEPARTMENT
```

More

To retrieve all the attributes, use * in SELECT clause

Retrieve all employees working for Dept. 5

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

Substring Comparisons, Arithmetic Operations, and Ordering

- like, binary operator for comparing strings
- %, wild card for strings
- _, wild card for characters
- ||, concatenate operation for strings

(name like '%a_') is true for all names having 'a' as second letter from the end.

- Partial strings are specified by using '

- SELECT FNAME, LNAME
- FROM EMPLOYEE
  WHERE FNAME LIKE '%Mc%';
- In order to list all employee who were born during 1960s we have the followings:
- SELECT FNAME, LNAME
- FROM EMPLOYEE
  WHERE BDATE LIKE '6_____';
- SQL also supports addition, subtraction, multiplication and division (denoted by +, , *, and /, respectively) on numeric values or attributes with numeric domains.

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

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

Retrieve all employees in department number 5 whose salary between $30000 and $40000.

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

It is possible to order the tuples in the result of a query.

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

The default order is in ascending order, but user can specify

ORDER BY DNAME DESC, LNAME ASC, FNAME, ASC;

Tables as Sets in SQL

SQL treats table as a **multiset,** which means duplicate tuples are OK SQL does not delete duplicate because Duplicate elimination is an expensive operation (sort and delete) user may be interested in the result of a query in case of aggregate function, we do not want to eliminate duplicates .To eliminate duplicate, use DISTINCT

examples

Q11: Retrieve the salary of every employee , and (Q!2) all distinct salary values

Q11: SELECT ALL SALARY

FROM EMPLOYEE

Q12: SELECT DISTINCT SALARY

FROM EMPLOYEE

Module 3

More Complex SQL Queries

Complex SQL queries can be formulated by composing nested SELECT/FROM/WHERE clauses within the WHEREclause of another query

Example: Q4: Make a list of Project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manger of the department that controls the project

```
Q4      SELECT DISTINCT PNUMBER
        FROM PROJECT
       WHERE PNUMBER IN (SELECT PNUMBER
                         FROM PROJECT, DEPARTMENT, EMPLOYEE
                        WHERE DNUM=DNUMBER AND MGRSSN=SSN AND
                       LNAME='Smith'
 OR   PNUMBER IN (SELECT PNO
                  FROM WORKS_ON, EMPLOYEE
                  WHERE ESSN=SSN AND LNAME='Smith')
```

IN operator and set of union compatible tuples

Example:

```
SELECT DISTINCT ESSN
FROM WORKS_ON
WHERE (PNO, HOURS) IN (SELECT PNO, HOURS
        FROM   WORKS_ON
        WHERE  SSN='123456789'
```

ANY, SOME and >, <=,<>,etc.

The keyword ALL

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v to a set of multiset V.

ALL V returns TRUE if v is greater than all the value in the set

Select the name of employees whose salary is greater than the salary of all the employees in department 5

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > ALL (SELECT SALARY
                    FROM EMPLOYEE
                    WHERE DNO=5);
```

Ambiguity in nested query

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

Correlated Nested Query

Whenever a condition in the WHEREclause of a nested query references some attributes of a relation declared in the outer query, the two queries are said to be correlated. The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) the outer query.

In general, any nested query involving the = or comparison operator IN can always be rewritten as a single block query

```
SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E, DEPENDENT D
WHERE E.SSN=D.ESSN AND E.SEX=D.SEX AND E.FNAME
=D.DEPENDENT=NAME
```

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

```
Q12:  SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE E.SSN IN (SELECT          ESSN
                      FROM   DEPENDENT
                      WHEREESSN=E.SSN AND
                      E.FNAME=DEPENDENT_NAME)
```

In Q12, the nested query has a different result for each tuple in the outer query.

The original SQL as specified for SYSTEM R also had a CONTAINS comparison operator, which is used in conjunction with nested correlated queries This operator was dropped from the language, possibly because of the difficulty in implementing it efficiently Most implementations of SQL do not have this operator The CONTAINS operator compares two sets of values , and returns TRUE if one set contains all values in the other set (reminiscent of the division operation of algebra).

Query 3: Retrieve the name of each employee who works on all the projects controlled by department number 5.

```
Q3: SELECT FNAME, LNAME
FROM EMPLOYEE WHERE ( (SELECT PNO FROM WORKS_ON WHERE SSN=ESSN)
CONTAINS (SELECT PNUMBER FROM PROJECT WHERE DNUM=5) )
```

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

THE EXISTS AND UNIQUE FUNCTIONS IN SQL

EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not We can formulate Query 12 in an alternative form that uses EXISTS as Q12B below.
Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

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

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

```
Q6:     SELECT  FNAME, LNAME
        FROM    EMPLOYEE
        WHERE NOT EXISTS   (SELECT    *
                            FROM   DEPENDENT
                            WHERESSN=ESSN)
```

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

EXPLICIT SETS AND NULLS IN SQL

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

Retrieve SSNs of all employees who work on project number 1,2,3

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

Null example

SQL allows queries that check if a value is NULL (missing or undefined or not applicable) SQL uses IS or IS NOT to compare NULLs because it considers each NULL value distinct from other NULL values, so equality comparison is not appropriate .

Retrieve the names of all employees who do not have supervisors

```
SELECT FNAME, LNAME
FROM EMPLOYEE
WHERE SUPERSSN IS NULL
```

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

Join Revisit

Retrieve the name and address of every employee who works for 'Search' department

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

Aggregate Functions

Include COUNT, SUM, MAX, MIN, and AVG

Query 15: Find the sum of the salaries of all employees the 'Research' dept, and the max salary, the min salary, and average:

```
SELECT SUM(SALARY), MAX(SALARY), MIN(SALARY) AVG(SALARY)
FROM EMPLOYEE
WHERE DNO=FNUMBER AND DNAME='RSEARCH'
```

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

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

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

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

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

Example of grouping

In many cases, we want to apply the aggregate functions to subgroups of tuples in a relation Each subgroup of tuples consists of the set of tuples that have the same value for the grouping attribute(s)

The function is applied to each subgroup independently

SQL has a GROUP BYclause for specifying the grouping attributes, which must also appear in the SELECTclause

For each project, select the project number, the project name, and the number of employees who work on that projet

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

In Q20, the EMPLOYEE tuples are divided into groupseach group having the same value for the grouping attribute DNO

The COUNT and AVG functions are applied to each such group of tuples separately.The SELECTclause includes only the grouping attribute and the functions to be applied on each group of tuples. A join condition can be used in conjunction with grouping

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

```
Q21:  SELECT    PNUMBER, PNAME, COUNT (*)
      FROM    PROJECT, WORKS_ON
      WHERE PNUMBER=PNO
      GROUP BY    PNUMBER, PNAME
```

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

THE HAVING CLAUSE:

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

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

```
Q22:    SELECT      PNUMBER, PNAME, COUNT (*)
        FROM    PROJECT, WORKS_ON
        WHERE PNUMBER=PNO
        GROUP BY    PNUMBER, PNAME
        HAVING      COUNT (*) > 2
```

SQL The Relational Database Standard

Update Statements in SQL

- **The Insert Command**

```
INSERT INTO EMPLOYEE  VALUES
(‘Richard’,’K’,’Marini’,653298653’,’30dec52’,98 Oak Forest, Katy,
TX’,’M’,37000,’987654321’,4)
```

More on Insert

Use explicit attribute names:

```
INSERT INTO EMPLOYEE (FNAME, LNAME,SSN)
VALUES (‘Richard’,’Marini’, ‘653298653’
```

- **The DELECT Command**

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

- **The UPDATE Command**

  Used to modify values of one or more selected tuples

  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;
```

Views in SQL

A view refers to a single table that is derived from other tables

```
CREATE VIEW WORKS_ON1
AS SELECT FNAME, LNAME, PNAME, HOURS
FROM EMPLOYEE, PROJECT, WORKS_ON WHERE SSN=ESSN AND PNO=PNUMBER
```

More on View

```
CREATE VIEW DEPT_INFO(DEPT_NAME, NO_OF_EMPLS, TOTAL_SAL)
AS SELECT DNAME, COUNT(*), SUM(SALARY)
FROM  DEPARTMENT, EMPLOYEE
WHERE DNUMBER=DNO
GROUP BY DNAME
```

More on view

Treat WORKS_ON1 like a base table as follows
SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNMAE='PROJECTX'

Main advantage of view:

Simplify the specification of commonly used queries
More on View
A View is always up to date;
A view is realized at the time we specify(or execute) a query on the view
DROP VIEW WORKS_ON1

Updating of Views

Updating the views can be complicated and ambiguous
In general, an update on a view on defined on a single table w/o any aggregate functions can be mapped to an update on the base table
More on Views
We can make the following observations:
A view with a single defining table is updatable if we view contain PK or CK of the base table
View on multiple tables using joins are not updatable
View defined using grouping/aggregate are not updatable

Specifying General Constraints

Users can specify certain constraints such as semantics constraints

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

Additional features

Granting and revoking privileges
Embedding SQL statements in a general purpose languages (C, C++, COBOL, PASCAL)
SQL can also be used in conjunction with a general purpose programming language, such as PASCAL, COBOL, or PL/I. The programming language is called the host language. The embedded SQL statement is distinguished from programming language statements by prefixing it with a special character or command so that a preprocessor can extract the SQL statements. In PL/I the keywords EXEC SQL precede any SQL statement. In some implementations, SQL statements are passed as parameters in procedure calls. We will use PASCAL as the host programming language, and a "$" sign to identify SQL statements in the program. Within an embedded SQL command, we may refer to program variables, which are prefixed by a "%" sign. The programmer should declare program variables to match the data types of the database attributes that the program will process.These program variables may or may not have names that are identical to their corresponding attributes.
Example: Write a program segment (loop) that reads a social security number and prints out some information from the corresponding EMPLOYEE tuple

```
E1:     LOOP:= 'Y';
        while LOOP = 'Y' do
                begin
                writeln('input social security number:');
```

```
readln(SOC_SEC_NUM);
$SELECT FNAME, MINIT, LNAME, SSN, BDATE,                    ADDR
  INTO %E.FNAME, %E.MINIT, %E.LNAME, %E.SSN,
                %E.BDATE, %E.ADDRESS, %E.SALARY
  FROM EMPLOYEE
  WHERE SSN=%SOC_SEC_NUM ;
writeln( E.FNAME, E.MINIT, E.LNAME, E.SSN,                  E.BDA
writeln('more social security numbers (Y or N)? ');
readln(LOOP)
end;
```

In E1, a single tuple is selected by the embedded SQL query; that is why we are able to assign its attribute values directly to program variables. In general, an SQL query can retrieve many tuples. The concept of a cursor is used to allow tupleatatime processing by the PASCAL programCURSORS: We can think of a cursor as a pointer that points to a single tuple (row) from the result of a query.The cursor is declared when the SQL query command is specified. A subsequent OPEN cursor command fetches the query result and sets the cursor to a position before the first row in the result of the query; this becomes the current row for the cursor. Subsequent FETCH commands in the program advance the cursor to the next row and copy its attribute values into PASCAL program variables specified in the FETCH command. An implicit variable SQLCODE communicates to the program the status of SQL embedded commands. An SQLCODE of 0 (zero) indicates successful execution. Different codes are returned to indicate exceptions and errors. A special END_OF_CURSOR code is used to terminate a loop over the tuples in a query result. A CLOSE cursor command is issued to indicate that we are done with the result of the query

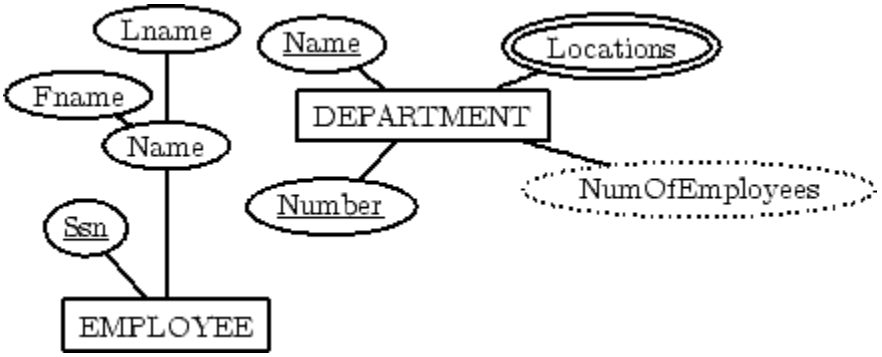
When a cursor is defined for rows that are to be updated the clause FOR UPDATE OF must be in the cursor declaration, and a list of the names of any attributes that will be updated follows.The condition WHERE CURRENT OF cursor specifies that the current tuple is the one to be updated (or deleted)

Example: Write a program segment that reads (inputs) a department name, then lists the names of employees who work in that department, one at a time. The program reads a raise amount for each employee and updates the employee's salary by that amount.

```
E2:     writeln('enter the department name:'); readln(DNAME);
        $SELECT DNUMBER INTO %DNUMBER
          FROM DEPARTMENT
          WHERE DNAME=%DNAME;
        $DECLARE EMP CURSOR FOR
                                SELECT SSN, FNAME, MINIT, LNAME, SALARY
                                FROM EMPLOYEE
                                WHERE DNO=%DNUMBER
          FOR UPDATE OF SALARY;
        $OPEN EMP;
        $FETCH EMP INTO %E.SSN, %E.FNAME, %E.MINIT, %E.LNAME, %E.SAL;
        while SQLCODE = 0 do begin
        writeln('employee name: ', E.FNAME, E.MINIT, E.LNAME);
        writeln('enter raise amount: ');  readln(RAISE);
        $UPDATE EMPLOYEE  SET SALARY = SALARY + %RAISE
        WHERE CURRENT OF EMP;
        $FETCH EMP INTO %E.SSN, %E.FNAME, %E.MINIT,
         %E.LNAME, %E.SAL;
        end;
        $CLOSE CURSOR EMP;
```

Relational Database Design Using ER-to-Relational Mapping

Step 1: For each **regular (strong) entity type** E in the ER schema, create a relation R that includes all the simple attributes of E.



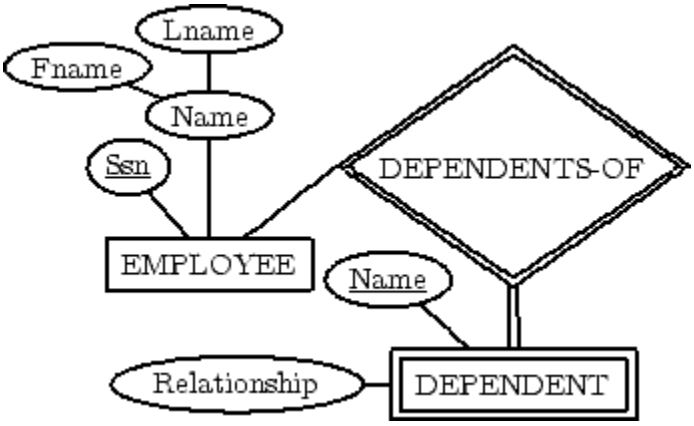
EMPLOYEE

SSN	Lname	Fname

DEPARTMENT

NUMBER	NAME

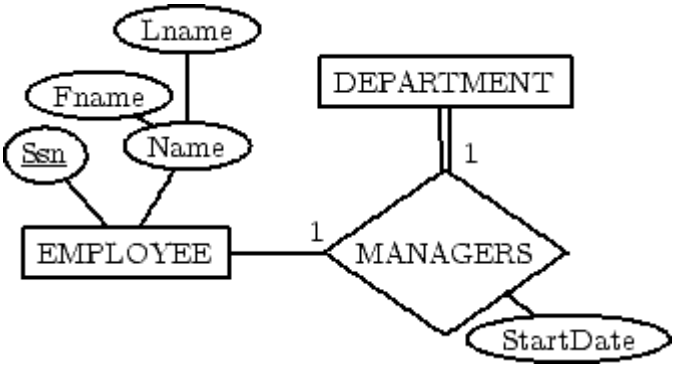
Step 2: For each **weak entity type** W in the ER schema with owner entity type E, create a relation R, and include all simple attributes (or simple components of composite attributes) of W as attributes. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).



DEPENDENT

EMPL-SSN	NAME	Relationship

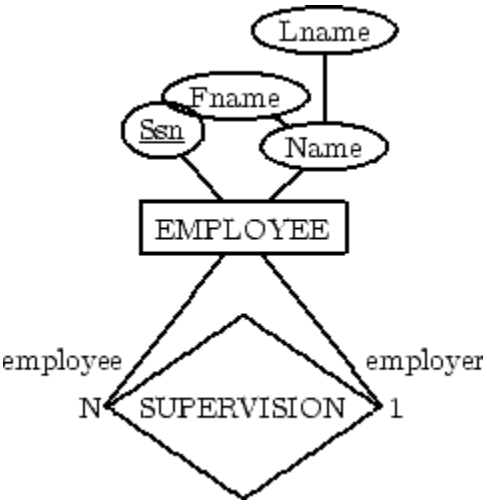
Step 3: For each **binary 1:1 relationship type** R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. Choose one of the relations, say S, and include the primary key of T as a foreign key in S. Include all the simple attributes of R as attributes of S.



DEPARTMENT

MANAGER-SSN	StartDate

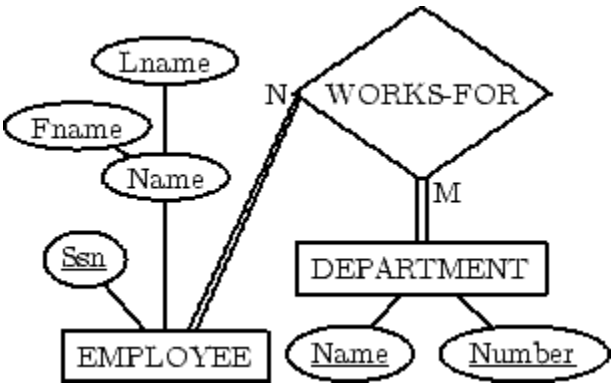
Step 4: For each regular **binary 1:N relationship type** R identify the relation (N) relation S. Include the primary key of T as a foreign key of S. Simple attributes of R map to attributes of S.



EMPLOYEE

SupervisorSSN

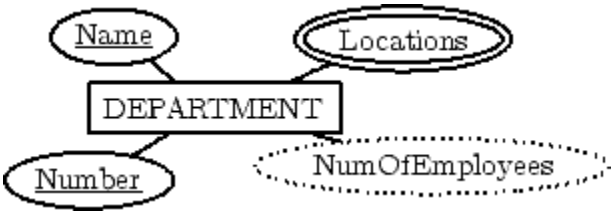
Step 5: For each **binary M:N relationship type** R, create a relation S. Include the primary keys of participant relations as foreign keys in S. Their combination will be the primary key for S. Simple attributes of R become attributes of S.



WORKS-FOR

EmployeeSSN	DeptNumber

Step 6: For each **multi-valued attribute A**, create a new relation R. This relation will include an attribute corresponding to A, plus the primary key K of the parent relation (entity type or relationship type) as a foreign key in R. The primary key of R is the combination of A and K.



DEP-LOCATION

Location	DEP-NUMBER

Step 7: For each **n-ary relationship type R**, where n>2, create a new relation S to represent R. Include the primary keys of the relations participating in R as foreign keys in S. Simple attributes of R map to attributes of S. The primary key of S is a combination of all the foreign keys that reference the participants that have cardinality constraint > 1.

For a recursive relationship, we will need a new relation.

Example: From Figure 3.2 to Figure 7.5

Summary

(Strong) Entity Type into Relation

- Include the simple attributes
- Include the simple components of the composite attributes
- Identify the primary keys

- Don't include: non-simple components of composite attributes, foreign keys, derived attributes, relational attributes

Weak Entity+Relationship Types into Relation

- Include simple attributes
- Add the owner's primary key attributes, as foreign key attributes
- Declare into a primary key the partial keys of the weak entity type combined with those imported from the owner

Binary 1:1 Relationship Types into Foreign Keys

- Include as foreign keys, in the relation of one entity type, the primary keys of the other entity type
- Include also the simple attributes of the relationship type
- If possible, the first entity type should have total participation in the relationship

Binary 1:N Relationship Types into Foreign Keys

- Add as foreign keys, to the relation of the entity type at the N side, the primary keys of the entity type at the 1 side
- Include also the simple attributes of the relationship type

Binary M:N Relationship Type into Relation

- Set as foreign keys the primary keys of the participating entity types
- Include the simple attributes of the relationship type

Multivalued Attribute into Relation

- Include the given attribute
- Include as foreign keys the primary attributes of the entity/relationship type owning the multivalued attribute

N-Ary Relationship Type

Similar to binary M:N relationship type

The Tuple Relational Calculus

- The tuple relational calculus is based on specifying a number of tuple variables.
- Each tuple variable usually *ranges over* a particular database relation.
- Relational calculus is a formal query language where we write one *declarative* expression to sepecify a retrieval request (with no description).

- In relational algebra we must write a *sequence of operations* to specify a retrieval request. That is, the relational algebra is a *procedural*, whereas the relational calculus is a *non-procedural* language.
- Any retrieval that can be specified in the relational algebra can also be specified in the relational calculus (and vice versa), that is, two languages are identical.
- A relational query language *L* is considered *relationally complete* if we can express in *L* any query that can be expressed in relational calculus.
- A simple tuple relational calculus query is of the form

  {t | COND(t)}

  for example,

  {t | EMPLOYEE(t) and t.SALARY > 50000}

  The above query retrieves all attribute values for each selected EMPLOYEE tuple t.

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

Tuple Variables and Range Relations

Examples

EMPLOYEE (FNAME, BINIT, LNAME, SSN, BDATE, ADDRESS, SEX, SALARY, SUPERSSN, DNO)

DEPT_LOC (DNUMBER, DLOCATION)

DEPARTMENT (DNAME, DNUMBER, MGRSSN, MGRSTARTDATE)

WORK_ON (ESSN, PNO, HOURS)

PROJECT (PNAME, PNUMBER, PLOCATION, DNUM)

DEPENDENT (ESSN, DEPENDENT_NAME, SEX, BDATE, RELATIONSHIP)

EMPLOYEE

Fname	Lname	SSN	SupervisorSSN

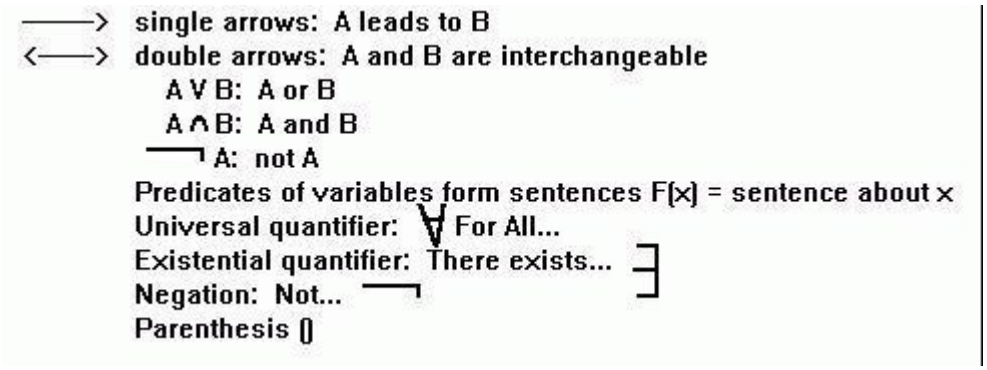
DEPARTMENT

NAME	NUMBER	MANAGER-SSN	StartDate

WORKS-FOR

EmployeeSSN	DeptNumber

The existential and Universal Quantifiers



For example in the following formulas:

F1 : d.NAME = `Research'

F2 : (∃t) (d.DNUMBER = t.DNO)

F3 : (∀d) (d.MGRSSN=`333445555')

The tuple variable *d* is free in both F1 and F2, whereas it is bound to the universal quantifier in F3.

Variable *t* is bound to the (∃) quantifier in F2.

A tuple variable *t* is bound if it is quantified. otherwise it is free.

Q1 (p 303) Retrieve the name and address of every employee who works for 'Research' department

{t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) and

(∃d) (DEPARTMENT(d) and d.DNAME='Research' and d.DNUMBER=t.DNO) }

t is the only free variable (in the left of midbar), but it is then bound successively to each tuple.

Q2 (p 304) For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

{p.NUMBER, p.DNUM, m.LNAME, m.BDATE, m.ADDRESS | PROJECT(p) and EMPLOYEE(m) and p.PLOCATION='Stafford' and

(∃ d)(DEPARTMENT(d) and p.DNUM = d.DNUMBER and d.MGRSSN=m.SSN) }

Q3' (p 304) Retrieve the name of each employee who works on some projects controlled by department number 5.

{e.LNAME, e.FNAME | EMPLOYEE(e) and (((∃ x)(∃ w)
(PROJECT(x))and WORKS_ON(w) and x.DNUM = 5) and w.ESSN=e.SSN and
x.PNUMBER=w.PNO)) }

Q3 (p 305) Retrieve the name of each employee who works on all the projects controlled by department number 5.

{e.LNAME, e.FNAME | EMPLOYEE(e) and (∀x) (not(PROJECT(x))or not (x.DNUM=5)or (∃w)(WORKS-ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO)) }

Find all employees which work in department #5.

{e.Fname, e.Lname | EMPLOYEE(e) and (∃ d ∃w)(DEPARTMENT(d) and WORKS-FOR(w) and w.EmployeeSSN = e.SSN and w.DeptNumber = 5) }

List the managers which have employees in all department.

{m.Fname, m.Lname | EMPLOYEE(m) and (∀ d ∃e ∃w)(not(DEPARTMENT(d)) or EMPLOYEE(e) and WORKS-FOR(w) and e.SupervisorSSN = m.SSN and e.SSN = w.EmployeeSSN and w.DeptNumber = d.NUMBER) }

Safe Expressions

- A *safe expression* in relational calculus is one that is guaranteed to yield a finite number of tuples as its result.
- The expression

  {t | not (EMPLOYEE(t))}

  is unsafe, because it yields all tuples in the universe that are not EMPLOYEE tuples.
- An expression is said to be safe if all values in its result are from the domain of the expression.

Quantifiers in SQL

- The *EXISTS* function in SQL is similar to the existential (∃) quantifier of the relational calculus.

- SELECT . . .
- FROM . . .
- WHERE EXISTS (SELECT *
- FROM R AS X
  WHERE P(X));
- SQL does not include a universal quantifier.
- NOT EXISTS in the SQL supports universal quantifier.

The Domain Relational Calculus

- The domain relational calculus was developed almost concurrently with SQL at IBM Research.
- The domain calculus differs from the tuple calculus in the type of variables used in formulas.
- To form a relation of degree *n* for a query result, we must have *n* domain variables (one for each attribute).

  For example,

  (EMPLOYEE)(qrstuvw) and q=`John' and r=`Smith') }

  Or alternatively we can write

  {tv | EMPLOYEE(`John', `Smith', q,r,s,t,u,v,w)}

Overview of the QBE Language

- The Query-By-Example (QBE) language is a graphical query language.
- It was developed at IBM Research and is available as an IBM commercial product as part of QMF (Query Management Facility).
- The query is formulated by filling in templates of relations that are displayed on a monitor screen.
- A domain variable is specified by the underscore character (_).
- A P. prefix denoteswe would like to print (or display) values in these columns.
- The constants specify values that must be exactly matched in these columns.

Module 4

Data Base design-1

4.1 Informal design guidelines for relation schemas

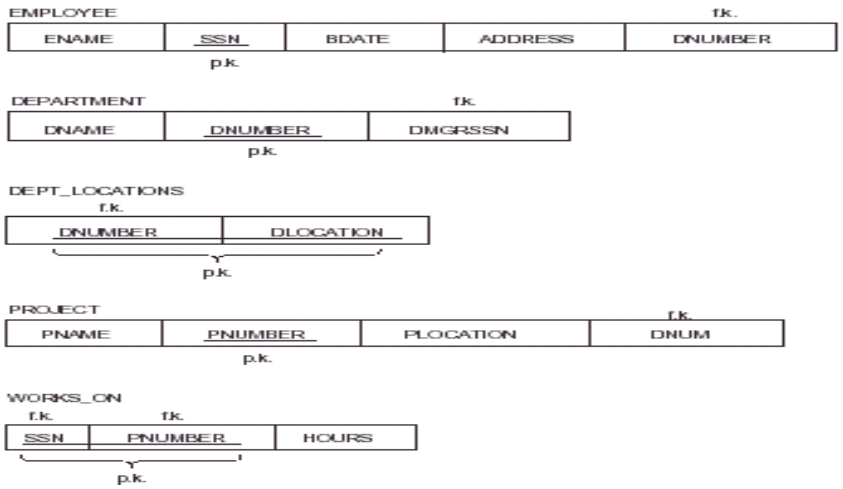
The four informal measures of quality for relation schema

- Semantics of the attributes
- Reducing the redundant values in tuples
- Reducing the null values in tuples
- Disallowing the possibility of generating spurious tuples

Semantics of relations attributes

Specifies how to interpret the attributes values stored in a tuple of the relation. In other words, how the attribute value in a tuple relate to one another.

Figure 14.1 Simplified version of the COMPANY relational database schema.

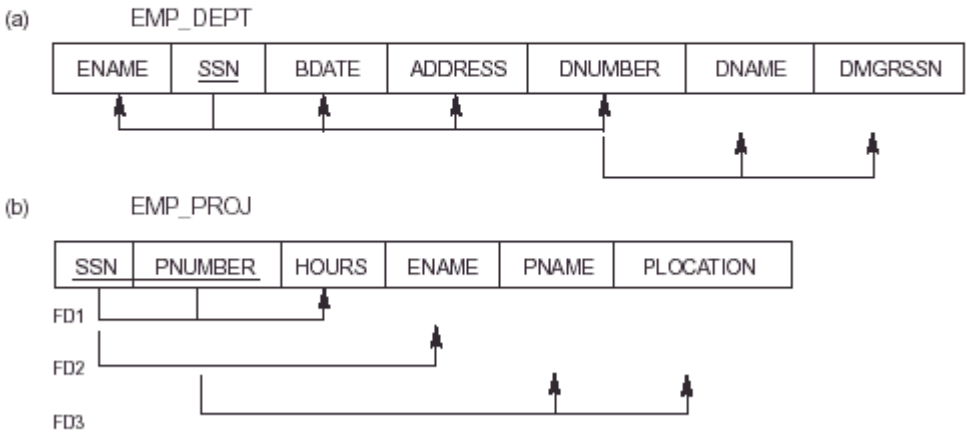


Guideline 1: Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation.

Reducing redundant values in tuples. Save storage space and avoid update anomalies.

- Insertion anomalies.
- Deletion anomalies.
- Modification anomalies.

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



Insertion Anomalies

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

It's difficult to insert a new department that has no employee as yet in the EMP_DEPT relation. The only way to do this is to place null values in the attributes for employee. This causes a problem because SSN is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity - not a department entity.

Deletion Anomalies

If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

Modification Anomalies

In EMP_DEPT, if we change the value of one of the attributes of a particular department- say the manager of department 5- we must update the tuples of all employees who work in that department.

Guideline 2: Design the base relation schemas so that no insertion, deletion, or modification anomalies occur. Reducing the null values in tuples. e.g., if 10% of employees have offices, it is

better to have a separate relation, EMP_OFFICE, rather than an attribute OFFICE_NUMBER in EMPLOYEE.

Guideline 3: Avoid placing attributes in a base relation whose values are mostly null. Disallowing spurious tuples.

Spurious tuples - tuples that are not in the original relation but generated by natural join of decomposed subrelations.

Example: decompose EMP_PROJ into EMP_LOCS and EMP_PROJ1.

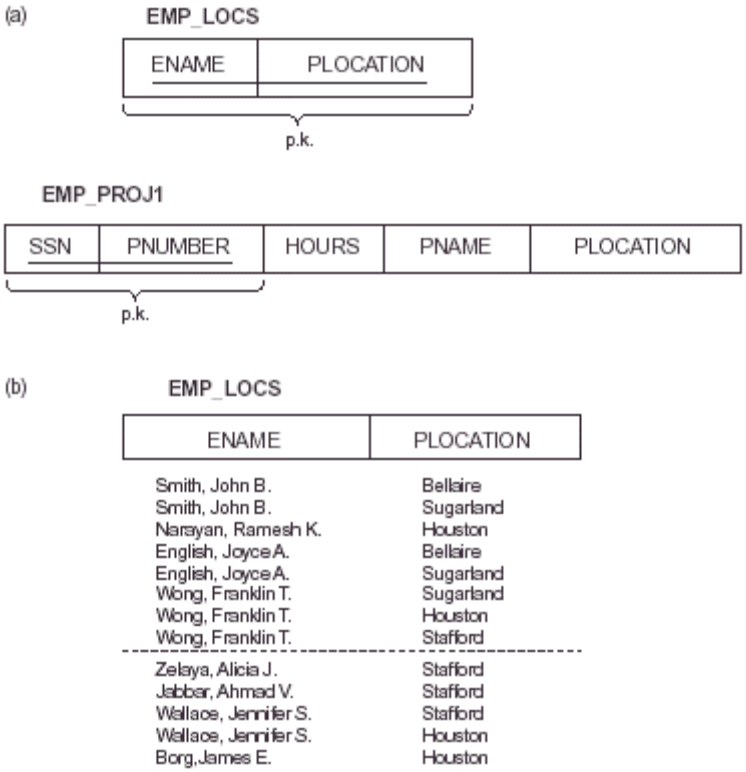


Fig. 14.5a

Guideline 4: Design relation schemas so that they can be naturally JOINed on primary keys or foreign keys in a way that guarantees no spurious tuples are generated.

A **functional dependency** (FD) is a constraint between two sets of attributes from the database. It is denoted by

$X \rightarrow Y$

We say that "*Y* is **functionally dependent** on *X*". Also, X is called the left-hand side of the FD. Y is called the right-hand side of the FD.

A functional dependency is a property of the semantics or meaning of the attributes, i.e., a property of the relation schema. They must hold on all relation states (extensions) of R. Relation extensions r(R). A FD X $\rightarrow Y$ is a full *functional dependency* if removal of any attribute from X means that the dependency does not hold any more; otherwise, it is a *partial functional dependency.*

Examples:

1. SSN $\rightarrow$ ENAME
2. PNUMBER $\rightarrow$ {PNAME, PLOCATION}
3. {SSN, PNUMBER} $\rightarrow$ HOURS

FD is property of the relation schema R, not of a particular relation state/instance

Let R be a relation schema, where X $\subseteq$ R and Y $\subseteq$ R

$t_1, t_2 \in r$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

The FD X $\rightarrow$ Y holds on R if and only if for all possible relations r(R), whenever two tuples of r agree on the attributes of X, they also agree on the attributes of Y.

- the single arrow $\rightarrow$ denotes "functional dependency"
- $X \rightarrow Y$ can also be read as "*X* determines *Y*"
- the double arrow $\Rightarrow$ denotes "logical implication"

4.2 Inference Rules

IR1. Reflexivity e.g. $X \rightarrow X$

- a formal statement of *trivial dependencies*; useful for derivations

IR2. Augmentation e.g. $X \rightarrow Y \Rightarrow XZ \rightarrow Y$

- if a dependency holds, then we can freely expand its left hand side

IR3. Transitivity e.g. $X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$

- the "most powerful" inference rule; useful in multi-step derivations

Armstrong inference rules are

sound

meaning that given a set of functional dependencies F specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds every relation state r of R that specifies the dependencies in F. In other words, rules can be used to derive precisely the closure or no additional FD can be derived.

complete

meaning that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of all possible dependencies that can be inferred from F. In other words, given a set of FDs, all implied FDs can be derived using these 3 rules.

Closure of a Set of Functional Dependencies

Given a set X of FDs in relation R, the set of all FDs that are implied by X is called the closure of X, and is denoted X^+.

Algorithms for determining X^+

$X^+ := X$;

repeat

old$X^+ := X^+$

for each FD $Y \rightarrow Z$ in F do

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

until old$X^+ = X^+$;

Example:

A $\rightarrow$ BC

E $\rightarrow$ CF

B $\rightarrow$ E

CD $\rightarrow$ EF

Compute $\{A, B\}^+$ of the set of attributes under this set of FDs.

Solution:

Step1: $\{A, B\}^+ := \{A, B\}$.

Go round the inner loop 4 time, once for each of the given FDs.

On the first iteration, for A $\rightarrow$ BC

$A \subseteq \{A, B\}^+$

$\{A, B\}^+ := \{A, B, C\}$.

Step2: On the second iteration, for $E \rightarrow CF$, $\notin \{A, B, C\}$

Step3 :On the third iteration, for $B \rightarrow E$

$B \subseteq \{A, B,C\}^+$

$\{A, B\}^+ := \{A, B, C, E\}$.

Step4: On the fourth iteration, for $CD \rightarrow EF$ remains unchanged.

Go round the inner loop 4 times again. On the first iteration result does not change; on the second it expands to {A,B,C,E,F}; On the third and forth it does not change.

Now go round the inner loop 4 times. Closure does not change and so the whole process terminates, with

$\{A,B\}^+ = \{A,B,C,E,F\}$

Example.

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

$\{SSN\}^+ = \{SSN, ENAME\}$

$\{PNUMBER\}^+ = ?$

$\{SSN,PNUMBER\}^+ = ?$

4.3 Normalization

The purpose of normalization.

- The problems associated with redundant data.
- The identification of various types of update anomalies such as insertion, deletion, and modification anomalies.
- How to recognize the appropriateness or quality of the design of relations.
- The concept of functional dependency, the main tool for measuring the appropriateness of attribute groupings in relations.
- How functional dependencies can be used to group attributes into relations that are in a known normal form.
- How to define normal forms for relations.
- How to undertake the process of normalization.

- How to identify the most commonly used normal forms, namely first (1NF), second (2NF), and third (3NF) normal forms, and Boyce-Codd normal form (BCNF).
- How to identify fourth (4NF), and fifth (5NF) normal forms.

Main objective in developing a logical data model for relational database systems is to create an accurate representation of the data, its relationships, and constraints. To achieve this objective, we must identify a suitable set of relations. A technique for producing a set of relations with desirable properties, given the data requirements of an enterprise

NORMAL FORMS

A relation is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes.

Any set of attributes of a relation schema is called a **superkey**. Every relation has at least one superkey—the set of all its attributes. A **key** is a *minimal superkey*, i.e., a superkey from which we cannot remove any attribute and still have the uniqueness constraint hold.

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key.** It is common to designate one of the candidate keys as the **primary key** of the relation. A **foreign key** is a key in a relation R but it's not a key (just an attribute) in other relation R' of the same schema.

Integrity Constraints

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.

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples of the two relations. Informally, the referential integrity constraint

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

An attribute of a relation schema R is called a **prime attribute** of the relation R if it is a member of *any key* of the relation R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

The goal of normalization is to create a set of relational tables that are free of redundant data and that can be consistently and correctly modified. This means that all tables in a relational database should be in the in the third normal form (3 NF).

Normalization of data can be looked on as a process during which unsatisfactory relation schemas are decomposed by breaking up their attributes into smaller relation schemas that possess desirable properties. One objective of the original normalization process is to ensure that the update anomalies such as insertion, deletion, and modification anomalies do not occur.

The most commonly used normal forms

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form

Other Normal Forms

- Fourth Normal Form
- Fifth Normal Form
- Domain Key Normal Form

First Normal Form (1NF)

First normal form is now considered to be part of the formal definition of a relation; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domains of attributes must include only atomic (simple,

indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute.

Practical Rule: "Eliminate Repeating Groups," i.e., make a separate table for each set of related attributes, and give each table a primary key.

Formal Definition: A relation is in first normal form (1NF) if and only if all underlying simple domains contain atomic values only.

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

(a)

(b) DEPARTMENT

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

(c) DEPARTMENT

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

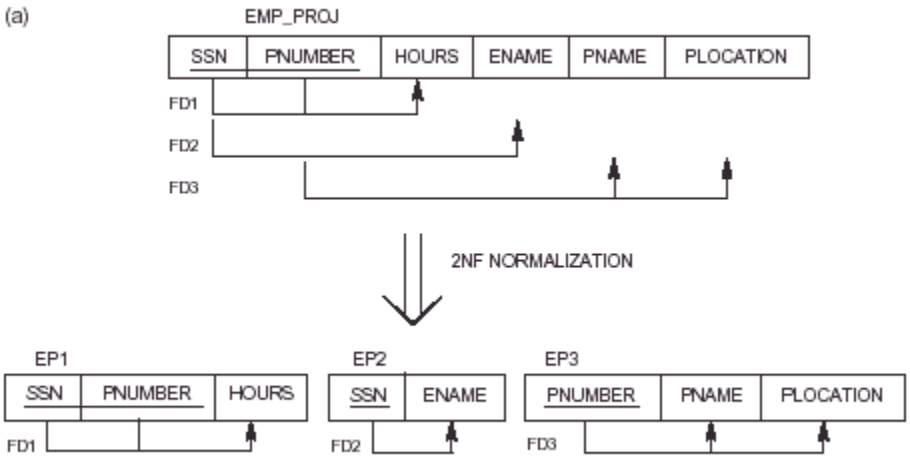
Second Normal Form (2NF)

Second normal form is based on the concept of fully functional dependency. A functional $X \rightarrow Y$ is a fully functional dependency is removal of any attribute A from X means that the dependency does not hold any more. A relation schema is in 2NF if every nonprime attribute in relation is fully functionally dependent on the primary key of the relation.

It also can be restated as: a relation schema is in 2NF if every nonprime attribute in relation is not partially dependent on any key of the relation.

Practical Rule: "Eliminate Redundant Data," i.e., if an attribute depends on only part of a multivalued key, remove it to a separate table.

Formal Definition: A relation is in second normal form (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key.



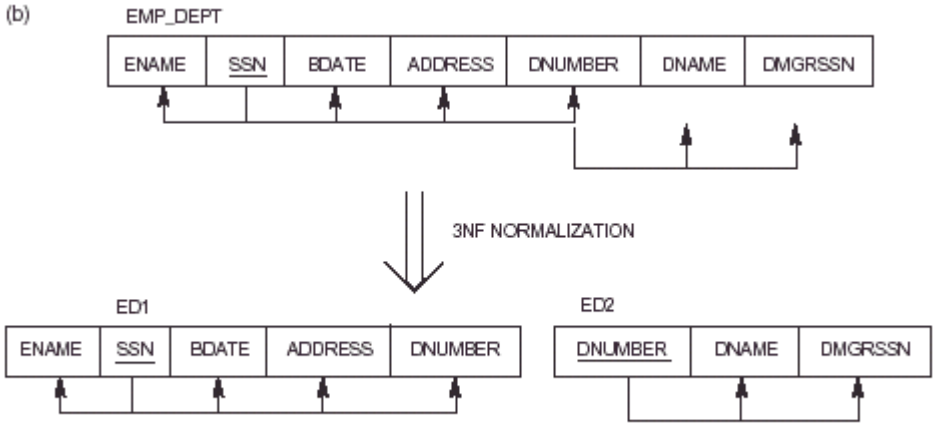
Third Normal Form (3NF)

Third normal form is based on the concept of transitive dependency. A functional dependency X→ Y in a relation is a transitive dependency if there is a set of attributes Z that is not a subset of any key of the relation, and both X→ Z and Z →Y hold. In other words, a relation is in 3NF if, whenever a functional dependency

X →A holds in the relation, either (a) X is a superkey of the relation, or (b) A is a prime attribute of the relation.

Practical Rule: "Eliminate Columns not Dependent on Key," i.e., if attributes do not contribute to a description of a key, remove them to a separate table.

Formal Definition: A relation is in third normal form (3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.



1NF: R is in 1NF iff all domain values are atomic.

2NF: R is in 2 NF iff R is in 1NF and every nonkey attribute is fully dependent on the key.

3NF: R is in 3NF iff R is 2NF and every nonkey attribute is non-transitively dependent on the key.

Boyce-Codd Normal Form (BCNF)

A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever a FD X -> A holds in R, then X is a superkey of R

- Each normal form is strictly stronger than the previous one:
- Every 2NF relation is in 1NF Every 3NF relation is in 2NF
- Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF

A relation is in BCNF, if and only if every determinant is a candidate key.

Additional criteria may be needed to ensure the the set of relations in a relational database are satisfactory.

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

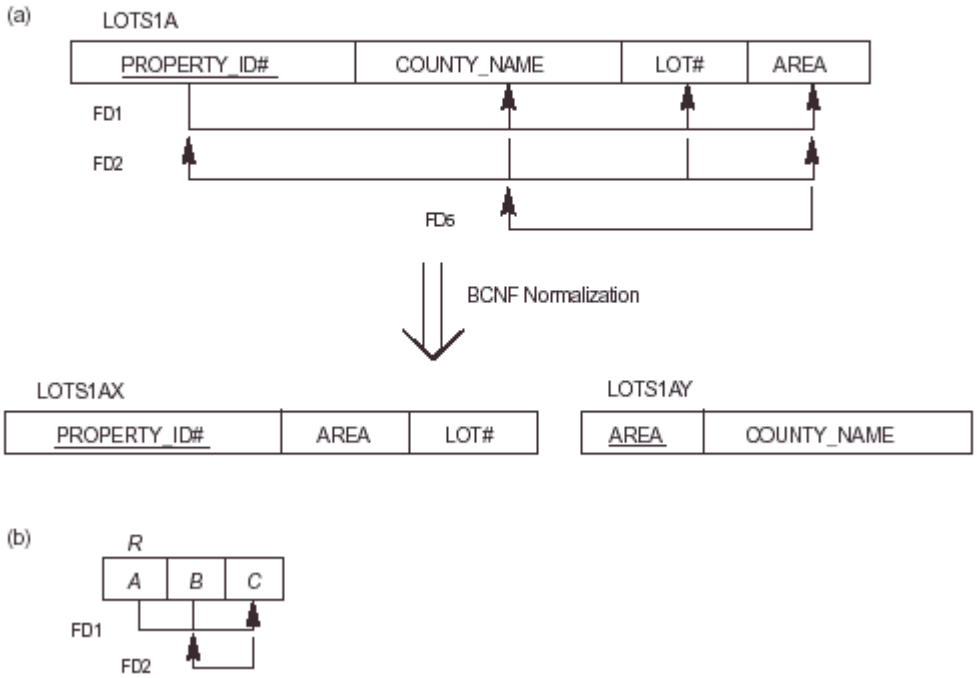


Figure 14.13 A relation TEACH that is in 3NF but not in BCNF.

TEACH

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

If X ⟶Y is non-trivial then X is a super key

STREET	CITY	ZIP

{CITY,STREET }⟶ ZIP

ZIP ⟶CITY

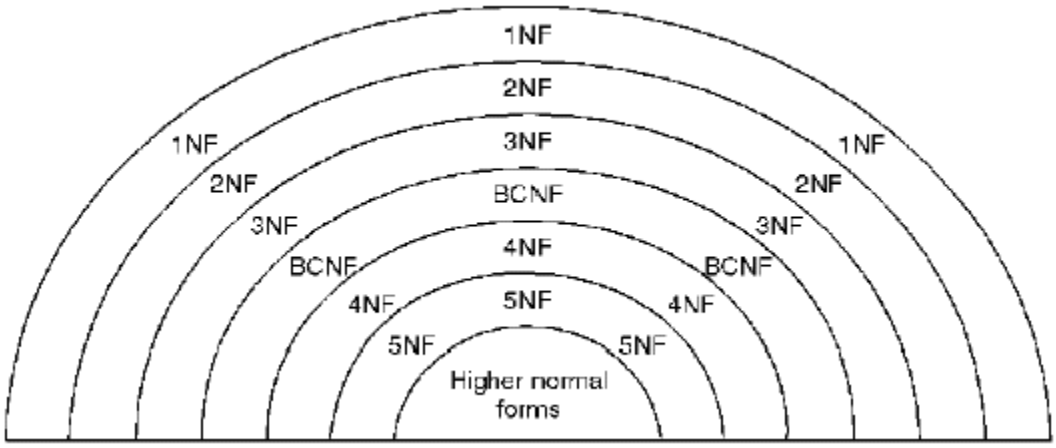
- Insertion anomaly: the city of a zip code can't be stored, if the street is not given

Normalization

STREET	ZIP

ZIP	CITY

Relationship Between Normal Forms



Normalization Algorithms based on FDs to synthesize 3NF and BCNF describe two desirable properties (known as properties of decomposition).

- Dependency Preservation Property
- Lossless join property

Dependency Preservation Property enables us to enforce a constraint on the original relation from corresponding instances in the smaller relations.

Lossless join property enables us to find any instance of the original relation from corresponding instances in the smaller relations (Both used by the design algorithms to achieve desirable decompositions).

A property of decomposition, which ensures that no spurious rows are generated when relations are reunited through a natural join operation.

Algorithms for Relational Database Schema Design

Individual relations being in higher normal do not guarantee a good deign Database schema must posses additional properties to guarantee a good design.

Relation Decomposition and Insufficiency of Normal Forms

Suppose R = { A_1, A_2, …, A_n} that includes all the attributes of the database. R is a universal relation schema, which states that every attribute name is unique. Using FDs, the algorithms decomposes the universal relation schema R into a set of relation schemas

D = {R_1, R_2, …, R_n} that will become the relational database schema; D is called a decomposition of R. Each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are lost; we have

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

This is called attribute preservation condition of a decomposition.

Decomposition and Dependency Preservation

We want to preserve dependencies because each dependencies in F represents a constraint on the database. We would like to check easily that updates to the database do not result in illegal relations being created.

It would be nice if our design allowed us to check updates without having to compute natural joins. To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.

Let F be a set of functional dependencies on schema R. Let D = {R1, R2, …, Rn} be a decomposition of R. Given a set of dependencies F on R, the projection of F on Ri, $\pi_{Ri}(F)$, where R_i is a subset of R, is the set of all functional dependencies $X \rightarrow Y$ such that attributes in $X \cup Y$ are all contained in Ri. Hence the projection of F on each relation schema Ri in the decomposition D is the set of FDs in F+, such that all their LHS and RHS attributes are in Ri. Hence, the projection of F on each relation schema Ri in the decomposition D is the set of functional dependencies in F+.

$((\pi_{R1}(F)) \cup (\pi_{R2}(F)) \cup \ldots \cup (\pi_{Rm}(F)))^+ = F^+$

i.e., the union of the dependencies that hold on each Ri belongs to D be equivalent to closure of F (all possible FDs)

/*Decompose relation, R, with functional dependencies, into relations, R_1,..., R_n, with associated functional dependencies,

F_1,..., F_k.

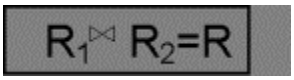
The decomposition is **dependency preserving if**:

If each functional dependency specified in F either appeared directly in one of the relation schema R in the decomposition D or could be inferred from the dependencies that appear in some R.

Lossless-join Dependency

A property of decomposition, which ensures that no spurious rows are generated when relations are reunited through a natural join operation.

Lossless-join property refers to when we decompose a relation into two relations - we can rejoin the resulting relations to produce the original relation.



Decompose relation, R, with functional dependencies, F, into relations, R1 and R2, with attributes, A1 and A2, and associated functional dependencies, F1 and F2.

- Decompositions are projections of relational schemas

R

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c3

$\pi_{A,B}$

A	B
a1	b1
a2	b2
a3	b1

$\pi_{B,C}$

B	C
b1	c1
b2	c2
b1	c3

- Old tables should be derivable from the newer ones through the natural join operation

$_{A,B}(R) \bowtie {}_{B,C}(R)$

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c3
a1	b1	c3
a3	b1	c1

- Wrong!
- R_1, R_2 is a lossless join decomposition of R iff the attributes common to R_1 and R_2 contain a key for at least one of the involved relations

R

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c1

$_{A,B}$

A	B
a1	b1
a2	b2
a3	b1

$_{B,C}$

B	C
b1	c1
b2	c2

- $_{A,B}(R) \cap {}_{B,C}(R) = B$

The decomposition is **lossless iff**:

- $A_1 \cup A_2 \cup A_1 \backslash A_2$ is in F^+, or
- $A_1 \cup A_2 \cup A_2 \backslash A_1$ is in F^+

However, sometimes there is the requirement to decompose a relation into more than two relations. Although rare, these cases are managed by join dependency and 5NF.

Multivalued Dependencies and Fourth Normal Form (4NF)

4NF associated with a dependency called **multi-valued dependency** (MVD). MVDs in a relation are due to first normal form (1NF), which disallows an attribute in a row from having a set of values.

MVD represents a dependency between attributes (for example, A, B, and C) in a relation, such that for each value of A there is a set of values for B, and a set of values for C. However, the set of values for B and C are independent of each other.

MVD between attributes A, B, and C in a relation using the following notation

Formal Definition of Multivalued Dependency

A **multivalued dependency** (MVD) X □□□□Y specified on R, where X, and Y are both subsets of R and Z = (R – (X Y)) specifies the following restrictions on r(R)

$t_3[X]=t_4[X]=t_1[X]=t_2[X]$

$t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$

$t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$

Fourth Normal Form (4NF)

A relation that is in Boyce-Codd Normal Form and contains no MVDs. BCNF to 4NF involves the removal of the MVD from the relation by placing the attribute(s) in a new relation along with a copy of the determinant(s).

A Relation is in 4NF if it is in 3NF and there is no multivalued dependencies.

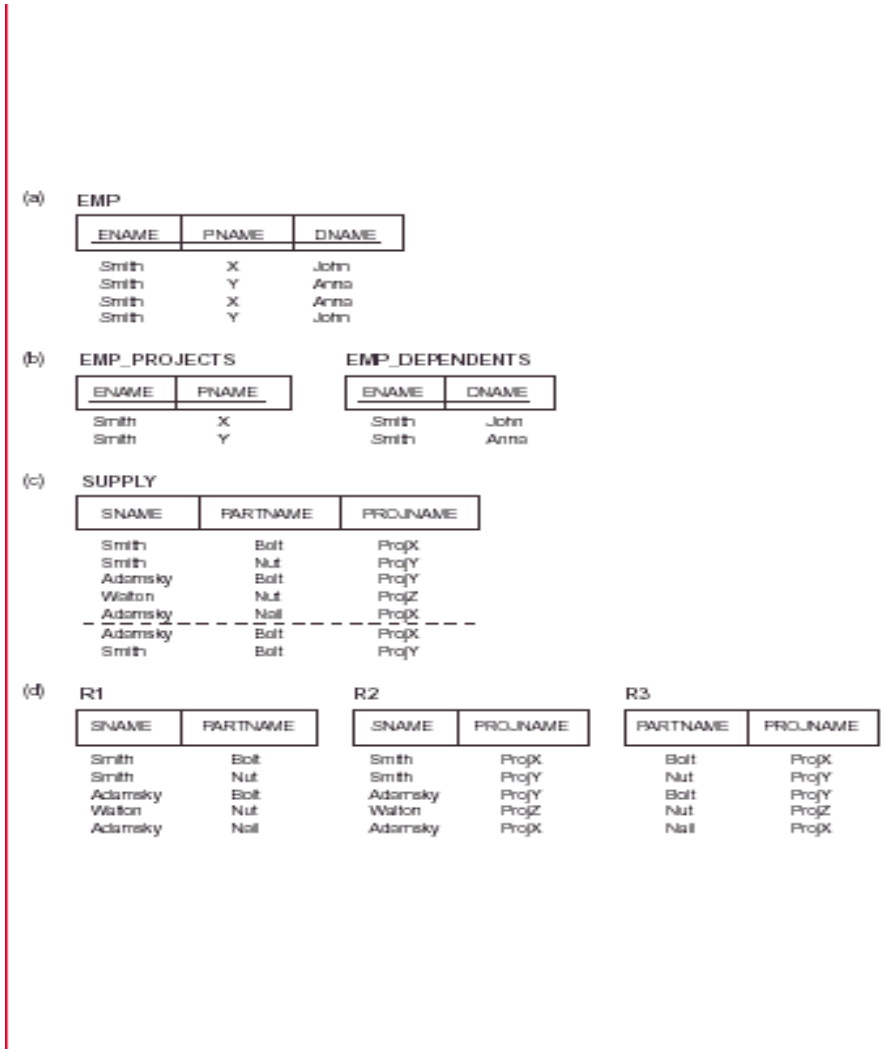


Figure 15.4 Fourth and fifth normal form. (a) The EMP relation with two MVDs: ENAME →→ PNAME and ENAME →→ DNAME. (b) Decomposing EMP into two relations in 4NF. (c) The relation SUPPLY with no MVDS satisfies 4NF but does not satisfy 5NF if the JD(R1, R2, R3) holds. (d) Decomposing SUPPLY into three 5NF relations.

Join Dependencies and 5 NF

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

* (π_{R1}(r), (π_{R2}(r), … (π_{Rn}(r)) = r

Lossless-join property refers to when we decompose a relation into two relations - we can rejoin the resulting relations to produce the original relation. However, sometimes there is the requirement to decompose a relation into more than two relations. Although rare, these cases are managed by join dependency and 5NF.

5NF (or project-join normal form (PJNF))

A relation that has no join dependency.

Template Dependencies

The idea behind template dependencies is to specify a template—or example—that defines each constraint or dependency. There are two types of templates: tuple-generating templates and constraint-generating templates. A template consists of a number of hypothesis tuples that are meant to show an example of the tuples that may appear in one or more relations. The other part of the template is the template conclusion. For tuple-generating templates, the conclusion is a set of tuples that must also exist in the relations if the hypothesis tuples are there. For constraint-generating templates, the template conclusion is a condition that must hold on the hypothesis tuples.

Domain Key Normal Form

The idea behind **domain-key normal form (DKNF)** is to specify (theoretically, at least) the "ultimate normal form" that takes into account all possible types of dependencies and constraints.

A relation is said to be in **DKNF** if all constraints and dependencies that should hold on the relation can be enforced simply by enforcing the domain constraints and key constraints on the relation.

However, because of the difficulty of including complex constraints in a DKNF relation, its practical utility is limited, since it may be quite difficult to specify general integrity constraints. For example, consider a relation CAR(MAKE, VIN#) (where VIN# is the vehicle identification number) and another relation MANUFACTURE(VIN#, COUNTRY) (where COUNTRY is the country of manufacture). A general constraint may be of the following form: "If the MAKE is either Toyota or Lexus, then the first character of the VIN# is a "*J*" if the country of manufacture is Japan; if the MAKE is Honda or Acura, the second character of the VIN# is a "*J*" if the country of manufacture is Japan." There is no simplified way to represent such constraints short of writing a procedure (or general assertions) to test them.

Module 5

Transaction Processing Concepts

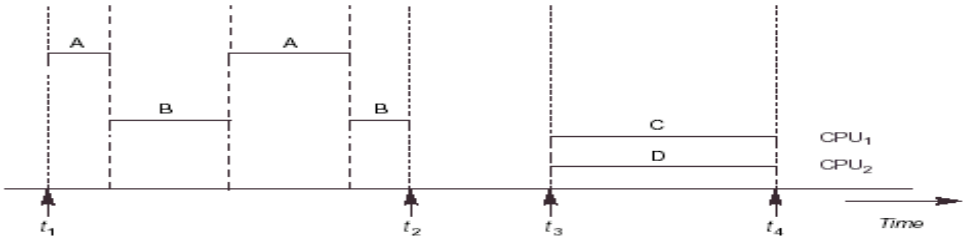
- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions
- Schedules and Recoverability
- Serializability of Schedules
- Transaction Support in SQL

Introduction to Transaction Processing

Single-User Versus Multiuser Systems

- A DBMS is **single-user** id at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.
- Most DBMS are multiuser (e.g., airline reservation system).
- *Multiprogramming operating systems* allow the computer to execute multiple programs (or processes) at the same time (having one CPU, concurrent execution of processes is actually interleaved).
- If the computer has multiple hardware processors (CPUs), *parallel processing* of multiple processes is possible.

Figure 19.1 Interleaved processing versus parallel processing of concurrent transactions.



Transactions, Read and Write Operations

- A *transaction* is a logical unit of database processing that includes one or more database access operations (e.g., insertion, deletion, modification, or retrieval operations). The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**
- *Read-only transaction* - do not changes the state of a database, only retrieves data.
- The basic database access operations that a transaction can include are as follows:
    - *read_item(X):* reads a database item *X* into a program variable *X*.
    - *write_item(X):* writes the value of program variable *X* into the database item named *X*.

Executing a read_item(*X*) command includes the following steps:

3. Find the address of the disk block that contains item *X*.
4. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
5. Copy item *X* from the buffer to the program variable named *X*.

Executing a write_item(*X*) command includes the following steps:

6. Find the address of the disk block that contains item *X*.
7. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
8. Copy item *X* from the program variable named *X* into its correct location in the buffer.
9. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Figure 19.2 Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

(a)

T_1
read_item (*X*);
X:=*X*-*N*;
write_item (*X*);
read_item (*Y*);
Y:=*Y*+*N*;
write_item (*Y*);

(b)

T_2
read_item (*X*);
X:=*X*+*M*;
write_item (*X*);

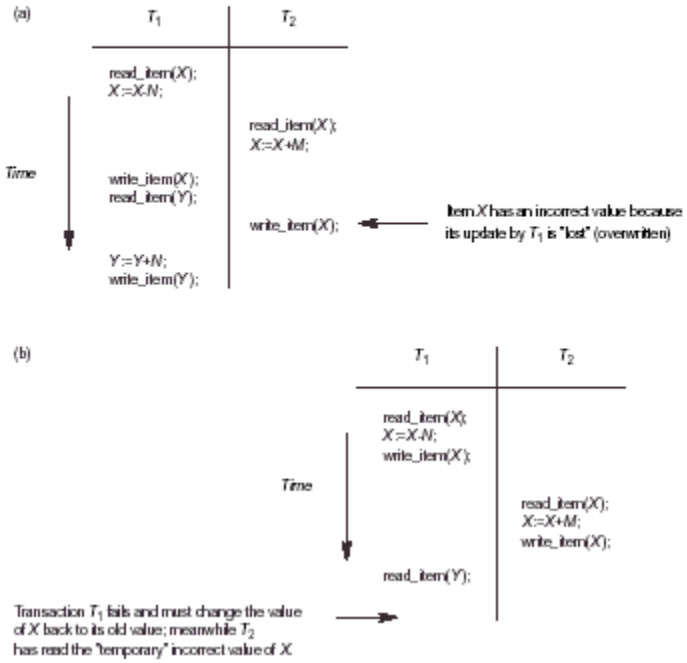
Why Concurrency Control Is Needed

- The Lost Update Problem.

  This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect. Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved then the final value of item *X* is incorrect, because T2 reads the value of *X before* T1 changes it in the database, and hence the updated value resulting from T1 is lost.

  For example, if *X* = 80 at the start (originally there were 80 reservations on the flight), *N* = 5 (T1 transfers 5 seat reservations from the flight corresponding to *X* to the flight corresponding to *Y*), and *M* = 4 (T2 reserves 4 seats on *X*), the final result should be *X* = 79; but in the interleaving of operations, it is *X* = 84 because the update in T1 that removed the five seats from *X* was *lost.*

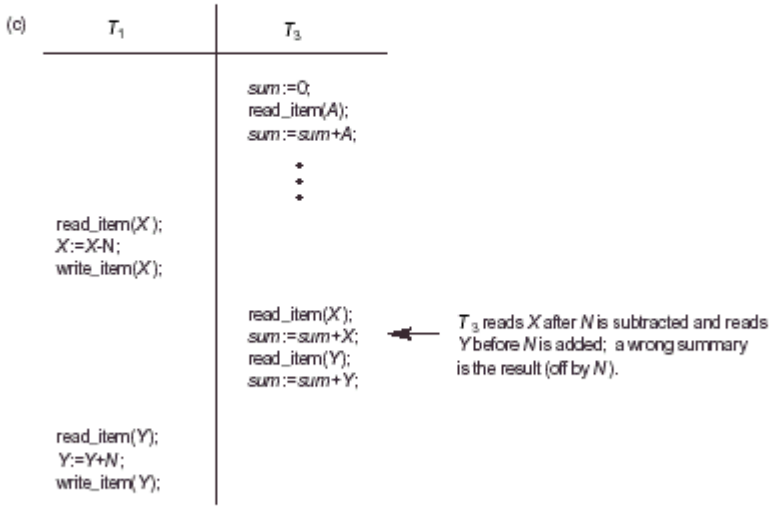
Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.



- **The Temporary Update (or Dirty Read) Problem.**

  This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value. Figure 19.03(b) shows an example where T1 updates item *X* and then fails before completion, so the system must change *X* back to its original value. Before it can do so, however, transaction T2 reads the "temporary" value of *X*, which will not be recorded permanently in the database because of the failure of T1. The value of item *X* that is read by T2 is called *dirty data*, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem.*

Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



- **The Incorrect Summary Problem.**

  If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure 19.03(c) occurs, the result of T3 will be off by an amount *N* because T3 reads the value of *X after N* seats have been subtracted from it but reads the value of *Y before* those *N* seats have been added to it.

  Another problem that may occur is called **unrepeatable read,** where a transaction *T* reads an item twice and the item is changed by another transaction *T'* between the two reads. Hence, *T* receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular

flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.

Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or (2) the transaction has no effect whatsoever on the database or on any other transactions. The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of *T* are not. This may happen if a transaction **fails** after executing some of its operations but before executing all of them.

Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

A computer failure (system crash): A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

A transaction or system error: Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error . In addition, the user may interrupt the transaction during its execution.

Local errors or exception conditions detected by the transaction: During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exception condition , such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.

Concurrency control enforcement: The concurrency control method (see Chapter 20) may decide to abort the transaction, to be restarted later, because it violates serializability (see Section 19.5) or because several transactions are in a state of deadlock.

Disk failure: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

Physical problems and catastrophes: This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

Transaction and System Concepts

Transaction States and Additional Operations

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (see below). Hence, the recovery manager keeps track of the following operations:

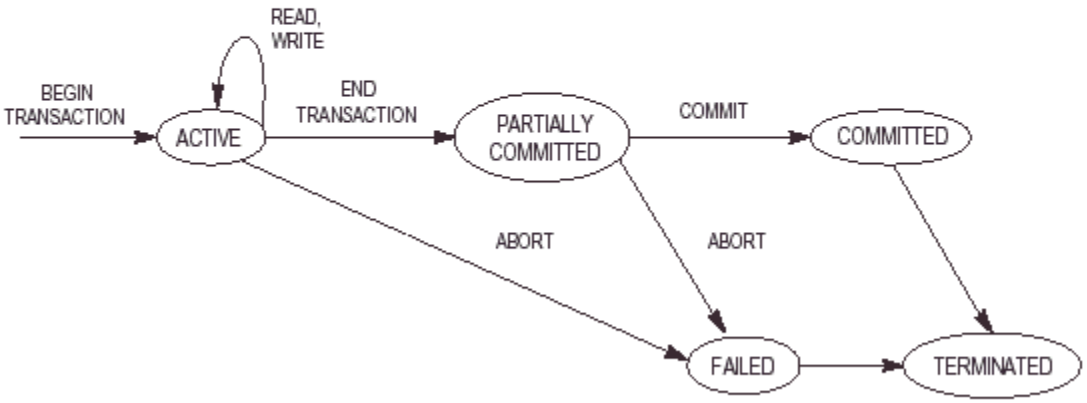
- BEGIN_TRANSACTION: This marks the beginning of transaction execution.
- READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.
- END_TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether

the transaction has to be aborted because it violates serializability or for some other reason.

- COMMIT_TRANSACTION: This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- ROLLBACK (or ABORT): This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

Figure shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can issue READ and WRITE operations. When the transaction ends, it moves to the **partially committed state.** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log). Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state.** Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

Figure 19.4 State transition diagram illustrating the states for transaction execution.



The System Log

- To be able to recover from failures that affect transactions, the system maintains a *log* to keep track of all transactions that affect the values of database items.
- Log records consists of the following information (*T* refers to a unique *transaction_id*):
    1. [start_transaction, *T*]: Indicates that transaction *T* has started execution.
    2. [write_item, *T,X,old_value,new_value*]: Indicates that transaction *T* has changed the value of database item *X* from *old_value* to *new_value.*
    3. [read_item, *T,X*]: Indicates that transaction *T* has read the value of database item *X.*
    4. [commit,*T*]: Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    5. [abort,*T*]: Indicates that transaction *T* has been aborted.

Desirable Properties of Transactions

Transactions should posses the following (ACID) properties:

Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

Schedules and Recoverability

A **schedule** (or **history**) *S* of *n* transactions T1, T2, ..., Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in *S*, the operations of Ti in *S* must appear in the same order in which they occur in Ti. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in *S*. For now, consider the order of operations in *S* to be a *total ordering*, although it is possible theoretically to deal with schedules whose operations form *partial orders*.

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

Similarly, the schedule for Figure 19.03(b), which we call Sb, can be written as follows, if we assume that transaction T1 aborted after its read_item(*Y*) operation:

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

1. they belong to different transactions;
2. they access the same item *X;* and
3. at least one of the operations is a write_item(*X*).

For example, in schedule S_a, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations w1(*X*) and w2(*X*). However, the operations r1(*X*) and r2(*X*) do not conflict, since they are both read operations; the operations w2(*X*) and w1(*Y*) do not conflict, because they operate on distinct data items *X* and *Y;* and the operations r1(*X*) and w1(*X*) do not conflict, because they belong to the same transaction.

A schedule *S* of *n* transactions T1, T2, ..., Tn, is said to be a **complete schedule** if the following conditions hold:

1. The operations in *S* are exactly those operations in T1, T2, ..., Tn, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction Ti, their order of appearance in *S* is the same as their order of appearance in Ti.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules Based on Recoverability

once a transaction *T* is committed, it should *never* be necessary to roll back *T*. The schedules that theoretically meet this criterion are called *recoverable schedules* and those that do not are called **nonrecoverable,** and hence should not be permitted.

A schedule *S* is recoverable if no transaction *T* in *S* commits until all transactions *T'* that have written an item that *T* reads have committed. A transaction *T* **reads** from transaction *T* in a schedule *S* if some item *X* is first written by T' and later read by *T*. In addition, T' should not have been aborted before *T* reads item *X*, and there should be no transactions that write *X* after T' writes it and before *T* reads it (unless those transactions, if any, have aborted before *T* reads *X).*

Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

S_a': $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

S_a' is recoverable, even though it suffers from the lost update problem. However, consider the two (partial) schedules S_c and S_d that follow:

S_c: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$
S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
S_e: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

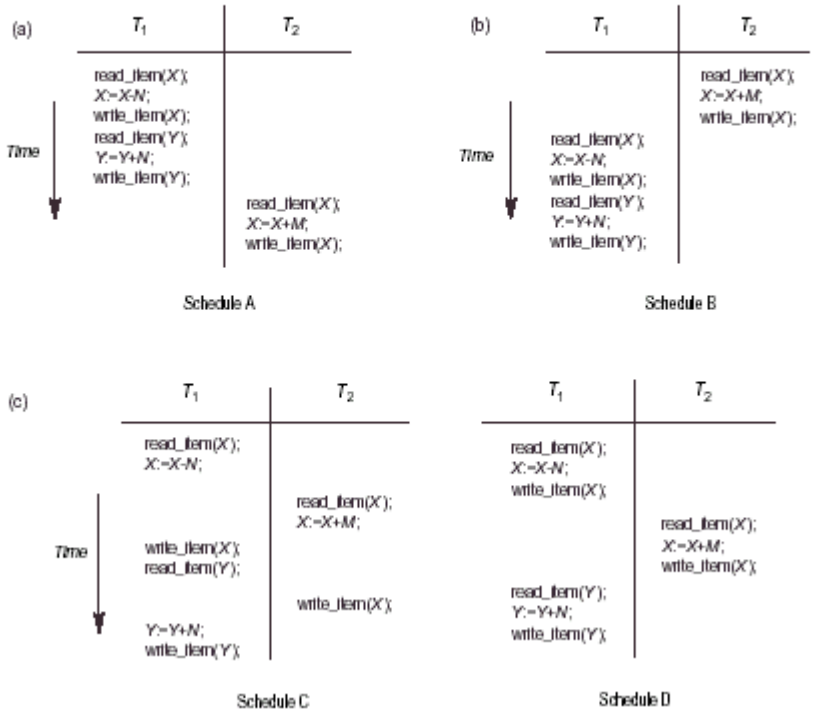
S_c is not recoverable, because T2 reads item *X* from T1, and then T2 commits before T1 commits. If T1 aborts after the c2 operation in S_c, then the value of *X* that T2 read is no longer valid and T2 must be aborted *after* it had been committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c2 operation in S_c must be postponed until after T1 commits. If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of *X* it read is no longer valid.

In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

Serializability of Schedules

- If no interleaving of operations is permitted, there are only two possible arrangement for transactions T1 and T2.
    1. Execute all the operations of T1 (in sequence) followed by all the operations of T2 (in sequence).
    2. Execute all the operations of T2 (in sequence) followed by all the operations of T1
- A schedule *S* is *serial* if, for every transaction *T* all the operations of *T* are executed consecutively in the schedule.
- A schedule *S* of *n* transactions is *serializable* if it is equivalent to some serial schedule of the same *n* transactions.

Figure 19.5 Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.



Transaction Support in SQL

- An SQL transaction is a logical unit of work (i.e., a single SQL statement).
- The *access mode* can be specified as *READ ONLY* or *READ WRITE*. The default is *READ WRITE*, which allows update, insert, delete, and create commands to be executed.

- The *diagnostic area size* option specifies an integer value *n*, indicating the number of conditions that can be held simultaneously in the diagnostic area.
- The *isolation level* option is specified using the statement *ISOLATION LEVEL*.
- the default isolation level is *SERIALIZABLE*.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
  READ WRITE
  DIAGNOSTICS SIZE 5
  ISOLATION LEVEL SERIALIZABLE;

EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
  VALUES ('Jabbar', 'Ahmad', '998877665', 2, 44000);
EXEC SQL UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1 WHERE DNO = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO:   EXEC SQL ROLLBACK;
THE_END:  . . . ;
```