MODULE 1
INTRODUCTION TO DATABASE

1.1 Introduction

Importance: Database systems have become an essential component of life in modern society, in that many frequently occurring events trigger the accessing of at least one database: bibliographic library searches, bank transactions, hotel/airline reservations, grocery store purchases, online (Web) purchases, etc., etc.

Traditional vs. more recent applications of databases:

The applications mentioned above are all "traditional" ones for which the use of rigidly-structured textual and numeric data suffices. Recent advances have led to the application of database technology to a wider class of data. Examples include **multimedia** databases (involving pictures, video clips, and sound messages) and **geographic** databases (involving maps, satellite images).

Also, database search techniques are applied by some WWW search engines.

Definitions

The term **database** is often used, rather loosely, to refer to just about any collection of related data. E&N say that, in addition to being a collection of related data, a database must have the following properties:

- It represents some aspect of the real (or an imagined) world, called the **miniworld** or **universe of discourse**. Changes to the miniworld are reflected in the database. Imagine, for example, a UNIVERSITY miniworld concerned with students, courses, course sections, grades, and course prerequisites.
- It is a logically coherent collection of data, to which some meaning can be attached. (Logical coherency requires, in part, that the database not be self-contradictory.)
- It has a purpose: there is an intended group of users and some preconceived applications that the users are interested in employing.

To summarize: a database has some source (i.e., the miniworld) from which data are derived, some degree of interaction with events in the represented miniworld (at least insofar as the data is updated when the state of the miniworld changes), and an audience that is interested in using it.

An Aside: data vs. information vs. knowledge: Data is the representation of "facts" or "observations" whereas information refers to the meaning thereof (according to some interpretation). Knowledge, on the other hand, refers to the ability to use information to achieve intended ends.

Computerized vs. manual: Not surprisingly (this being a CS course), our concern will be with computerized database systems, as opposed to manual ones, such as the card catalog-based systems that were used in libraries in ancient times (i.e., before the year 2000). (Some authors wouldn't even recognize a non-computerized collection of data as a database, but E&N do.)

Size/Complexity: Databases run the range from being small/simple (e.g., one person's recipe database) to being huge/complex (e.g., Amazon's database that keeps track of all its products, customers, and suppliers).

Definition: A **database management system** (DBMS) is a collection of programs enabling users to create and maintain a database.

More specifically, a DBMS is a *general purpose* software system facilitating each of the following (with respect to a database):

- **definition:** specifying data types (and other constraints to which the data must conform) and data organization
- **construction:** the process of storing the data on some medium (e.g., magnetic disk) that is controlled by the DBMS
- **manipulation:** querying, updating, report generation
- **sharing:** allowing multiple users and programs to access the database "simultaneously"
- **system protection:** preventing database from becoming corrupted when hardware or software failures occur
- **security protection:** preventing unauthorized or malicious access to database.

Given all its responsibilities, it is not surprising that a typical DBMS is a complex piece of software.

A database together with the DBMS software is referred to as a **database system**. (See Figure 1.1, page 7.)

1.2 : An Example:

UNIVERSITY database in Figure 1.2. Notice that it is relational!

Among the main ideas illustrated in this example is that each file/relation/table has a set of named fields/attributes/columns, each of which is specified to be of some data type. (In addition to a data type, we might put further restrictions upon a field, e.g., GRADE_REPORT must have a value from the set {'A', 'B', ..., 'F'}.)

The idea is that, of course, each table will be populated with data in the form of records/tuples/rows, each of which represents some entity (in the miniworld) or some relationship between entities.

For example, each record in the **STUDENT** table represents a —surprise!— student. Similarly for the **COURSE** and **SECTION** tables.

On the other hand, each record in **GRADE_REPORT** represents a relationship between a student and a section of a course. And each record in **PREREQUISITE** represents a relationship between two courses.

Database *manipulation* involves *querying* and *updating*.

Examples of (informal) queries:

- Retrieve the transcript(s) of student(s) named 'Smith'.
- List the names of students who were enrolled in a section of the 'Database' course in Spring 2006, as well as their grades in that course section.
- List all prerequisites of the 'Database' course.

Examples of (informal) updates:

- Change the CLASS value of 'Smith' to sophomore (i.e., 2).
- Insert a record for a section of 'File Processing' for this semester.
- Remove from the prerequisites of course 'CMPS 340' the course 'CMPS 144'.

Of course, a query/update must be conveyed to the DBMS in a precise way (via the query language of the DBMS) in order to be processed.

As with software in general, developing a new database (or a new application for an existing database) proceeds in phases, including **requirements analysis** and various levels of **design** (conceptual (e.g., Entity-Relationship Modeling), logical (e.g., relational), and physical (file structures)).

1.3 : Characteristics of the Database Approach:

Database approach vs. File Processing approach: Consider an organization/enterprise that is organized as a collection of departments/offices. Each department has certain data processing "needs", many of which are unique to it. In the **file processing approach**, each department would control a collection of relevant data files and software applications to manipulate that data.

For example, a university's Registrar's Office would maintain data (and programs) relevant to student grades and course enrollments. The Bursar's Office would maintain data (and programs) pertaining to fees owed by students for tuition, room and board, etc. (Most likely, the people in these offices would not be in direct possession of their data and programs, but rather the university's Information Technology Department would be responsible for providing services such as data storage, report generation, and programming.)

One result of this approach is, typically, **data redundancy**, which not only wastes storage space but also makes it more difficult to keep changing data items consistent with one another, as a change to one copy of a data item must be made to all of them (called **duplication-of-effort**). **Inconsistency** results when one (or more) copies of a datum are changed but not others. (E.g., If you change your address, informing the Registrar's Office should suffice to ensure that your grades are sent to the right place, but does not guarantee that your next bill will be, as the copy of your address "owned" by the Bursar's Office might not have been changed.)

In the **database approach**, a single repository of data is maintained that is used by all the departments in the organization. (Note that "single repository" is used in the logical sense. In physical terms, the data may be *distributed* among various sites, and possibly *mirrored*.)

Main Characteristics of database approach:

1. **Self-Description:** A database system includes —in addition to the data stored that is of relevance to the organization— a complete definition/description of the database's structure and constraints. This **meta-data** (i.e., data about data) is stored in the so-called **system catalog**, which contains a description of the structure of each file, the type and storage format of each field, and the various constraints on the data (i.e., conditions that the data must satisfy).

   See Figures 1.1 and 1.3.

   The system catalog is used not only by users (e.g., who need to know the names of tables and attributes, and sometimes data type information and other things), but also by the DBMS software, which certainly needs to "know" how the data is structured/organized in order to interpret it in a manner consistent with that structure. Recall that a DBMS is *general purpose*, as opposed to being a specific database application. Hence, the structure of the data cannot be "hard-coded" in its programs (such as is the case in typical *file processing* approaches), but rather must be treated as a "parameter" in some sense.

2. **Insulation between Programs and Data; Data Abstraction:**

   Program-Data Independence: In traditional file processing, the structure of the data files accessed by an application is "hard-coded" in its source code. (E.g., Consider a file descriptor in a COBOL program: it gives a detailed description of the layout of the records in a file by describing, for each field, how many bytes it occupies.)

   If, for some reason, we decide to change the structure of the data (e.g., by adding the first two digits to the YEAR field, in order to make the program Y2K compliant!), **every** application in which a description of that file's structure is hard-coded must be changed!

In contrast, DBMS access programs, in most cases, do not require such changes, because the structure of the data is described (in the system catalog) separately from the programs that access it and those programs consult the catalog in order to ascertain the structure of the data (i.e., providing a means by which to determine boundaries between records between fields within records) so that they interpret that data properly See Figure 1.4.

In other words, the DBMS provides a conceptual or logical view of the data to application programs, so that the underlying implementation may be changed without the programs being modified. (This is referred to as *program-data independence*.)
Also, which access paths (e.g., indexes) exist are listed in the catalog, helping the DBMS to determine the most efficient way to search for items in response to a q uery.

Data Abstraction:

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

Note: In fairness to COBOL, it should be pointed out that it has a COPY feature that allows different application programs to make use of the same file descriptor stored in a "library". This provides some degree of program-data independence, but not nearly as much as a good DBMS does. *End of note.*

Example by which to illustrate this concept: Suppose that you are given the task of developing a program that displays the contents of a particular data file. Specifically, each record should be displayed as follows:

```
Record #i:
  value of first field
  value of second field
  ...
  ...
  value of last field
```

To keep things very simple, suppose that the file in question has fixed-length records of 57 bytes with six fixed-length fields of lengths 12, 4, 17, 2, 15, and 7 bytes, respectively, all of which are ASCII strings. Developing such a program would not be difficult. However, the obvious solution would be tailored specifically for a file having the particular structure described here and would be of no use for a file with a different structure.

Now suppose that the problem is generalized to say that the program you are to develop must be able to display *any* file having fixed-length records with fixed-length fields that are ASCII strings. Impossible, you say? Well, yes, unless the program has the ability to access a description of the file's structure (i.e., lengths of its records and the fields

therein), in which case the problem is not hard at all. This illustrates the power of metadata, i.e., data describing other data.

3. **Multiple Views of Data:** Different users (e.g., in different departments of an organization) have different "views" or perspectives on the database. For example, from the point of view of a Bursar's Office employee, student data does not include anything about which courses were taken or which grades were earned. (This is an example of a **subset** view.)

   As another example, a Registrar's Office employee might think that GPA is a field of data in each student's record. In reality, the underlying database might calculate that value each time it is needed. This is called **virtual** (or **derived**) data.

   A view designed for an academic advisor might give the appearance that the data is structured to point out the prerequisites of each course.

   (See Figure 1.5, page 14.)

   A good DBMS has facilities for defining multiple views. This is not only convenient for users, but also addresses security issues of data access. (E.g., The Registrar's Office view should not provide any means to access financial data.)

4. **Data Sharing and Multi-user Transaction Processing:** As you learned about (or will) in the OS course, the simultaneous access of computer resources by multiple users/processes is a major source of complexity. The same is true for multi-user DBMS's.

   Arising from this is the need for **concurrency control**, which is supposed to ensure that several users trying to update the same data do so in a "controlled" manner so that the results of the updates are as though they were done in some sequential order (rather than interleaved, which could result in data being incorrect).

   This gives rise to the concept of a **transaction**, which is a process that makes one or more accesses to a database and which must have the appearance of executing in *isolation* from all other transactions (even ones that access the same data at the "same time") and of being *atomic* (in the sense that, if the system crashes in the middle of its execution, the database contents must be as though it did not execute at all).

   Applications such as **airline reservation systems** are known as **online transaction processing** applications.

1.4 : Actors on the Scene

These apply to "large" databases, not "personal" databases that are defined, constructed, and used by a single person via, say, Microsoft Access.

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

1. **Database Administrator** (DBA): This is the chief administrator, who oversees and manages the database system (including the data and software). Duties include authorizing users to access the database, coordinating/monitoring its use, acquiring hardware/software for upgrades, etc. In large organizations, the DBA might have a support staff.
2. **Database Designers**: They are responsible for identifying the data to be stored and for choosing an appropriate way to organize it. They also define **views** for different categories of users. The final design must be able to support the requirements of all the user sub-groups.
3. **End Users**: These are persons who access the database for **querying**, **updating**, and **report generation**. They are main reason for database's existence!
   - **Casual end users:** use database occasionally, needing different information each time; use query language to specify their requests; typically middle- or high-level managers.
   - **Naive/Parametric end users**: Typically the biggest group of users; frequently query/update the database using standard **canned transactions** that have been carefully programmed and tested in advance. Examples:
     - bank tellers check account balances, post withdrawals/deposits
     - reservation clerks for airlines, hotels, etc., check availability of seats/rooms and make reservations.
     - shipping clerks (e.g., at UPS) who use buttons, bar code scanners, etc., to update status of in-transit packages.
   - **Sophisticated end users**: engineers, scientists, business analysts who implement their own applications to meet their complex needs.
   - **Stand-alone users**: Use "personal" databases, possibly employing a special-purpose (e.g., financial) software package. Mostly maintain personal databases using ready-to-use packaged applications.
   - An example is a tax program user that creates its own internal database.
   - Another example is maintaining an address book
4. **System Analysts, Application Programmers, Software Engineers**:
   - **System Analysts**: determine needs of end users, especially naive and parametric users, and develop specifications for canned transactions that meet these needs.
   - **Application Programmers**: Implement, test, document, and maintain programs that satisfy the specifications mentioned above.

1.5: Workers Behind the Scene

DBMS system designers/implementors: provide the DBMS software that is at the foundation of all this!

- **tool developers**: design and implement software tools facilitating database system design, performance monitoring, creation of graphical user interfaces, prototyping, ets.
- **operators and maintenance personnel**: responsible for the day-to-day operation of the system.

1.6: Capabilities/Advantages of DBMS's

1. **Controlling Redundancy:** Data redundancy (such as tends to occur in the "file processing" approach) leads to **wasted storage space**, **duplication of effort** (when multiple copies of a datum need to be updated), and a higher liklihood of the introduction of **inconsistency**.

   On the other hand, redundancy can be used to improve performance of queries. Indexes, for example, are entirely redundant, but help the DBMS in processing queries more quickly.

   Another example of using redundancy to improve performance is to store an "extra" field in order to avoid the need to access other tables (as when doing a JOIN, for example). See Figure 1.6 (page 18): the StudentName and CourseNumber fields need not be there.

   A DBMS should provide the capability to automatically enforce the rule that no inconsistencies are introduced when data is updated. (Figure 1.6 again, in which Student_name does not match Student_number.)

2. **Restricting Unauthorized Access:** A DBMS should provide a **security and authorization subsystem**, which is used for specifying restrictions on user accounts. Common kinds of restrictions are to allow read-only access (no updating), or access only to a subset of the data (e.g., recall the Bursar's and Registrar's office examples from above).
3. **Providing Persistent Storage for Program Objects:** Object-oriented database systems make it easier for complex runtime objects (e.g., lists, trees) to be saved in secondary storage so as to survive beyond program termination and to be retrievable at a later time.
4. **Providing Storage Structures for Efficient Query Processing:** The DBMS maintains indexes (typically in the form of trees and/or hash tables) that are utilized to improve the execution time of queries and updates. (The choice of which indexes to create and maintain is part of *physical database design and tuning* (see Chapter 16) and is the responsibility of the DBA.

   The **query processing and optimization** module is responsible for choosing an efficient query execution plan for each query submitted to the system. (See Chapter 15.)

5. **Providing Backup and Recovery:** The subsystem having this responsibility ensures that recovery is possible in the case of a system crash during execution of one or more transactions.

   Providing Multiple User Interfaces: For example, query languages for casual users, programming language interfaces for application programmers, forms and/or command codes for parametric users, menu-driven interfaces for stand-alone users.

6. **Representing Complex Relationships Among Data:** A DBMS should have the capability to represent such relationships and to retrieve related data quickly.
7. **Enforcing Integrity Constraints:** Most database applications are such that the semantics (i.e., meaning) of the data require that it satisfy certain restrictions in order to make sense. Perhaps the most fundamental constraint on a data item is its data type, which specifies the universe of values from which its value may be drawn. (E.g., a Grade field could be defined to be of type Grade_Type, which, say, we have defined as including precisely the values in the set { "A", "A-", "B+", ..., "F" }.

   Another kind of constraint is *referential integrity*, which says that if the database includes an entity that refers to another one, the latter entity must exist in the database. For example, if (R56547, CIL102) is a tuple in the Enrolled_In relation, indicating that a student with ID R56547 is taking a course with ID CIL102, there *must be* a tuple in the Student relation corresponding to a student with that ID.

8. **Permitting Inferencing and Actions Via Rules:** In a **deductive** database system, one may specify *declarative* rules that allow the database to infer new data! E.g., Figure out which students are on academic probation. Such capabilities would take the place of application programs that would be used to ascertain such information otherwise.

   Active database systems go one step further by allowing "active rules" that can be used to initiate actions automatically.

1.7 : A Brief History of Database Applications

- Early Database Applications:
  - The Hierarchical and Network Models were introduced in mid 1960s and dominated during the seventies.
  - A bulk of the worldwide database processing still occurs using these models.
- Relational Model based Systems:
  - Relational model was originally introduced in 1970, was heavily researched and experimented with in IBM Research and several universities.
  - Object-oriented and emerging applications:

Object-Oriented Database Management Systems (OODBMSs) were introduced in late 1980s and early 1990s to cater to the need of complex data processing in CAD and other applications.

- Their use has not taken off much.

Many relational DBMSs have incorporated object database concepts, leading to a new category called *object-relationa*l DBMSs (ORDBMSs)

Extended relational systems add further capabilities (e.g. for multimedia data, XML, and other data types)

- Relational DBMS Products emerged in the 1980s
- Data on the Web and E-commerce Applications:
- Web contains data in HTML (Hypertext markup language) with links among pages.
- This has given rise to a new set of applications and E-commerce is using new standards like XML (eXtended Markup Language).
- Script programming languages such as PHP and JavaScript allow generation of dynamic Web pages that are partially generated from a database
  - New functionality is being added to DBMSs in the following areas:
  - Scientific Applications
  - XML (eXtensible Markup Language)
  - Image Storage and Management
  - Audio and Video data management
  - Data Warehousing and Data Mining
  - Spatial data management
  - Time Series and Historical Data Management
  - The above gives rise to *new research and development* in incorporating new data types, complex data structures, new operations and storage and indexing schemes in database systems.
  - Also allow database updates through Web pages

1.8: 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.

ENTITY-RELATIONSHIP MODEL

2.1 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.)

2.1.2: 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.

2.2 DBMS Architecture and Data Independence

2.2.1: Three-Schema Architecture: (See Figure 2.2, page 34.) 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 > 3.5.

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.

2.3 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.)

2.3.1 DBMS Languages

menu-based, forms-based, gui-based, natural language, special purpose for parametric users, for DBA.

2.3.2 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

2.4 Database System Environment

See Figure 2.3, page 41.

2.5 Centralized and Client/Server Architectures for DBMS's

2.6 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

2.7 Data Modeling Using the Entity-Relationship Model

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.

2.8 : 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**.

2.8.1 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 3.4, 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 tree (as in the aforementioned Figure 3.4). 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.

A more complicated example of a complex attribute is *AddressPhone* in Figure 3.5 (page 65). This attribute is for recording data regarding addresses and phone numbers of a business. 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.

2.8.2 : 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. (An example of this appears in Figure 3.7, page 67, in which 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

2.8.3 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*

2.9 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):

- EMPLOYEEMANAGESDEPARTMENT (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)
- EMPLOYEESUPERVISESEMPLOYEE (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 **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). See Figure 3.9 on page 71 for a graphical depiction of this.

2.9.1 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.

2.9.2 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*.

2.9.3 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:

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

  $\{ (a_1, b_1), (a_2, b_2), \ldots (a_m, b_m) \}$,

  then (at that same moment) *A*'s instance set must be $\{ a_1, a_2, \ldots, a_m \}$. 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.

2.9.4 Attributes of Relationship Types (page 76)

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.

2.10 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).

Questions

1. Design an ER Diagram for keeping track of Information about Bank Database,Taking into account 4 entities?
2. Describe how to map the following Scenario‘s in ER Model to schema,with suitable exam ple:
3. List the summary of the notations for ER diagrams. Include symbols used in ER diagram and their meaning.
4. With respect to ER model explain with example.
5. What is meant by partial key? Explain.
6. Define an entity and an attribute,explain the different types of attributes that occur in an ER diagram model,with an example
7. Define the following with an example
    i. Weak entity types
    ii. Cardinality ratio
    iii. Ternary relationship
    iv. Participation constraints