Chapter I: Introduction to Database

0.1 Some Definitions:

- Database is a collection of related data.
- Data is facts that can be recorded and have implicit meaning.
- Database Management System (DBMS) is a computerized system that enables users to create and maintain a database.
- DBMS is a **general-purpose software system** that faciliates the processes of *defining, constructing, manipulating* and *sharing* databases among various users and applications.

0.2 DBMS System:

- **Defining the database** involves specify the data types, structures, and constraints of the data to be stored in the database.
- **Meta-data**, The database definition or descriptive information is also stored by the DBMS in the form of a database catalog and dictionary.
- **Constructing the database** is the process of storing the data on some storage medium that is controlled by the DBMS.
- **Manipulating the database** includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld.
- **Sharing the database** allows multiple users and programs to access the database simultaneously.

0.3 Application Program:

- An **application program** accesses the database by sending queries or requests for data to DBMS.
- A **query** typically causes data to be retrieved.
- A **transaction** causes data to be read and written into the database.

- DBMS also provides functions for **protecting** and **maintaining** the database system:

- **Protecting** includes *system protection* against hardware or software malfuncton (or crash) and *security protection* against unauthorized and malicious access.
- A DBMS need to **maintain** the database system by allowing the system as requirement change over-time.

Database system = database + DBMS

- Conceptual Design.
- Logic Design.
- Physical Design.

- Design of the new application for an existing database or design of a brand new database starts off with a phase called **requirements specification and analysis**.

- These requirements are documented in detail and transformed into a **conceptual design** that can be represented and manipulated by some computerized tools → easily modified, maintained and transformed into **database implementation**.

- The design is then translated into **logical design**. that can be expressed in a data model implemented in a commercial DBMS.

- The final stage is called **physical design**, further specifications are provided for storing and access database.

0.4 Characteristic of the Database Approach:

1. File processing:
   - A traditional **file processing**, each user defines and implements the files needed for the specific software application.
   - Both users are interested in same data but each users maintain separate files and programs to manipulate files.
   - This redundancy in defining and storing data results in wasted storage space and in redundant effort to maintain common up-to-date data.
2. Database Approach:
   - In database approach, a single repository maintains data that defined once and then accessed by various users repeatedly though queries, transactions, and application programs.
   - The main characteristic of the database approach vs file processing:
     - Self-describing nature of a database system.
     - Insulation (ngăn cách) between programs and data, and data abstraction.
     - Support of the multiple views of the data.
     - Sharing of data and multiuser transaction processing.

   (a) Self-Describing Nature of a Database System:
   - The database system contain not only the database itself but also a complete definition or description of database structure and constraints.

- The information stored in the DBMS catalog is called **meta-data**, it describes the structure of the primary database.
- NOSQL systems do not require meta-data, those data is stored in **self-describing data** that includes the data item names and data values together in 1 structure.
- The DBMS software must work equally well with *any number of database applications*

(b) Insulation between Programs and Data, and Data Abstraction:

- The structure of data files is stored in the DBMS catalog separately from the access program, this is called **program-data independence**.
  (Why **meta-data enable data-program independence ?**)
- An operation (known as function or method) is specific in *interface* and *implementation*.
  - interface includes operation name and its data types of its argument.
  - implementation of the operation is specified separately and can be changed without affecting the interface.
- This may be known as **program-operation independence**.
- The characteristic that allows program-data independence and program-operation independence is called **data abstraction**.
- A **data model** is a type of data abstraction that is used to provided this conceptual representation. The data model *hides* storage and implementation out of database users.

(c) Support of multiple views of the Data:

- A multiuser DBMS whose users have a variety of distinct applications must provide facilities for multiple views.

(d) Sharing of Data and Multiuser Transaction Processing:

- This is essential if data for multiple users to be integrated and maintained for a single database.
- The DBMS must include **concurrency control** software to ensure that several users trying to update the same data so that the results of update is correct.

3. Actors on the Scene:

- Database Administrators.
- Database Designers.
- End users.

0.5 Advantages of Using the DBMS Approach

1. Reduce Redundancy:
   - Data normalization: All logical data item are stored in *only one place* in the database.
2. Restricting Unauthorized Access:
3. Provide persistent storage for Program Object:
4. Provide Storage Structures and Search Techniques for efficient Query processing:
5. Provide Backup and Recovery:
6. Provide Multiple User Interfaces:
7. Represent Complex Relationships among Data:
8. Enforcing Integrity Constraints:
9. Permitting Interfencing and Actions using Rules and Triggers:

0.6 When not to use DBMS

1. It maybe more desirable to develop customized database applications:
    - Simple, well-defined database applications that are not expected to change at all.
    - Stringent, real-time requirements for some application programs that may not met base of DBMS overhead.
    - Embedded devices with limited storage capacity, where a general-purpose DBMS would not fit.
    - No multiple user-access data.

Chapter II: Database System Concepts and Architecture

0.7 Data Models, Schemas, and Instances

1. Data Models:
    - A data model is a collection of concepts that can be used to describe the structure of the database - provides the necessary means to achieve this abstraction.
    - Most data models also include set of **basic operations** for specifying retrievals and updates on the database.
    - Types of Data Model:
        - **High-level** or **conceptual data models** provide concepts that are close to the way many users realize data.
        - **Low-level** or **physical data model** provide concepts that describe the detail of how data is stored in the computer storage media, typically magnetic disk.
        - **Representational** (or **implementation**) **data models** provide concepts that may be easily understood by end users but that are not too far removed (very different from) from the way data is organized in computer storage.
2. Schemas, Instances, and Database State:
    - Schemas is kind of layout of the database.
    - The actual data in the database may change quite frequently. The data in the database at the particular moment of time is called a **database state** or **snapshot**

0.8 Three-Schema Architecture and Data Independence

1. The three-schema Architecture:
    - The **internal level** has an **internal schema** describes the physical storage structure of the database.
    - The **conceptual level** has a **conceptual schema** describes the structure of the whole database for users.

- The **external** or **view level** includes a number of **external schemas** or **user views**.

2. Data Independence:

   (a) **Logical data independence:** is the capacity to change the conceptual schema without having to change external schemas or application programs.

   (b) **Physical data independence:** is the capacity to change the internal schema without having to change the conceptual schema

   (c) Summary: Data independence occurs when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, plication referring to the higher level schema do not need to be changed.

0.9 Database Languages and Interfaces

1. DBMS Languages:

   - **Data definition language** (DDL) : used to define both conceptual and external schemas where no strict separation of levels is maintained.
   - **Storage definition language** (SDL):
     - is used to specify the internal schema.
     - DDL is used in conceptual schema.
     - It is used where there is a clear separation between the conceptual and external levels
   - **View definition language** (VDL) : is used to specify user views and their mapping to the conceptual and external schemas.
   - Beside, DBMS also provides a set of operations or a language called the **data manipulation language** (DML) for manipulating the database.

     - A **high-level** or **nonprocedural** DML.
       * It is used to specify complex database operations concisely (chính xác).
       * Many DBMSs allow high-level DML statements to be entered from the display monitor or terminal or to be embedd in a general-purpose programming language.
       * High-level DML, such as SQL can specify and retrieve many records in a single DML statement, it is called **set-at-a-time**
     - A **low-level** or **procedural** DML.
       * It *must* be embedded in a general-purposes language.
       * It is also called as **record-at-a-time** since it functions related to retrieving and processing individuals records and objects.
   - Whenever DMLS commands, whether high-level or low-level, are embedded in a general-purpose language and that language is called the **host language** and DML is called the **data sub language**.
   - A high-level DML used in a standalone interactive manner is called a **query language**.

2. DBMS Interfaces:

   - Menu-based Interfaces for Web Clients or Browsing

- Apps for Mobile Devices
- Forms-based Interfaces
- Graphic User Interfaces
- Natural Language Interfaces
- Keyboard-based Database Search
- Search Input and Output

0.10 The Database System Environment

1. **DBMS Component Modules**
   - Many DBMSs have **buffer management** module to schedule disk read/write since management of buffer storage has a considerable effect on performance by reducing disk read/write.
   - A high-level **sorted data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.
2. **Database System Utilities**
   - Most DBMSs have **database utilities** to help the DBA manage the database system.
     - Loading
     - Backup
     - Database storage reorganization
     - Performance monitoring (monitors database and provide video statistics to the DBA)

End

Chapter III: Conceptual Data Modeling and Database Design

"The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints, using the concept provided by high-level data model."

"ER model is a logical organization of data within a database system. ER model technique is based on relational data model"

0.11 Entity Types, Entity Sets, Attributes, and Keys

1. **Entity**
    - Entity is a *thing* or *object* in the real world with an independence existence.
    - An entity may be an object with a physical existence or it may be an object with a conceptual existence (Ex: company, university jobs, etc).
2. **Attribute**
    - Each entity has attributes - the particular properties that describe it.
    - Types of attributes in the ER model:
        - **Composite versus Simple**
            - **Composite attributes** can be divided into smaller subparts, which represent more **basic** attributes with independence meanings.
            - Attributes that are not divisible are called **simple** or **atomic attributes**
        - **Single-Valued versus Multivalued Attributes**
            - Attributes have a single value for a particular entity are called **single valued entity**.
            - Attributes have a multiple value for a particular entity are called **multivalued entity**.
        - **Stored versus Derived Attributes**
            - Attributes that are related to or can be determined by another attributes are called **derived attributes**.
            - The others are called **stored attributes**.
        - **NULL Values**
            - Attributes that particular entities may not have an applicable value for. (*not applicable*)
            - NULL can be also be used if we do not know the value of an attribute for a particular entity. (*unknown*)
        - **Complex Attributes**: composite and multivalued can be nested arbitrarily.
3. **Entity Types and Entity Sets**
    - An **entity type** defines a *collection* (or *set*) of entities that have the same attributes.
    - Each entity type in the database is described by its name and attributes.
    - The collection of all entities of a particular entity type in the database at any point is called an **entity set** or **entity collection**.
    - The entity set is usually referred to using the same name as the entity type, even though they are two separate concepts.

- An entity type describes the **schema** or **intension** for *a set of entities* that share the same structure.
- The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

4. **Key Attributes of an Entity Type**
   - An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes.
   - An entity type has one or more attributes whose values are distinct for each individual entity in the entity set, those attributes are called **key attributes**, and its value can be used to identify each entity uniquely.
   - An attribute is a key of an entity type means that the preceding uniqueness property must hold for *every entity set* of the entity type.
5. **Value Sets (Domains) of Attributes**
   - Each simple attribute of an entity type is associated with a **value set** (or **domain** of the values), which specifies the set of values that may be assigned to that attribute for each individual entity.

0.12 Relationship Types, Relationship Sets, Roles, and Structural Constraints

1. Relationship Types, Sets, and Instances
   - **Relationship Types, Sets, and Instances**
     - A **relationship type** R among n entity types $E_1, E2, \ldots, E_n$ defines a set of associations - or a **relationship set** - among entities from these entity types.
     - The relationship set R is a set of **relationship instances** r_i, where each r_i associates n individual entities $(e_1, e_2, \ldots, e_n)$ and each entity e_j in r_i is a member of entity set E_j.
   - **Relationship Degree, Role Names, and Recursive Relationships**
     - **Degree of a Relationship Type**
       - The **degree** of a relationship type is the number of participating entity type.
     - **Relationship as Attributes**
     - **Role Names and Recursive Relationships**
       - The **role name** signifies the role that a participating entity from the entity type plays in each role relationship instance, and it helps to explain what the relationship means.
       - Role name are not technically necessary in relationship types where all the participating entity are distinct, since each participating entity type name can by used as the role name.
       - In case the *same* entity type participates in a relationship type in *different roles*, the role name becomes essential.
         → Such relationship types are called **recursive relationship** or **self-referencing relationships**.
   - **(Structural) Constraints on Binary Relationships Types**
     - **Cardinality Ratios for Binary Relationships**
       - The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate on.
     - **Participation Constraints and Existence Dependencies**

        * The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type.
        * This constraint specifies the *minimum* number of relationship instances that each entity can participate in, called the **minimum cardinality constraint**.
        * There are two types of participation constraints:
            - **Total participation:** every entity in the total set of entities must related to another entity set via a relationship, called **existence dependency**
            - **Partial Participation:** some or *some or part of the set of* entities are related to some other entities via a relationship.
- **Attributes of Relationship Types**
    - Relationship types can also have attributes, similar to those of entity types.
    - The attributes of 1:1 relationship types can be determined separately, as either one of the participating types.
    - However, for 1:N relationship types, a relationship attribute can be migrated *only* to the entity type on the N-side of the relationship.
    - For M:N (many-to-many) relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes must be *specified as relationship attributes*.

0.13 Weak Entity Types

1. **Definitions:**

    - Entity types that do not have key attributes of their own are called **weak entity types**.
    - In contrast, **regular entity types** that do have a key attributes are called **strong entity types**.
    - Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. Those entities are called **identifying** or **owner entity type**.
    - Relationship type that relates to a weak entity type to its owner is called **identifying relationship** of the weak entity type.
    - A weak entity type always have a *total participation constraint* (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity.
    - A weak entity type normally has a **partial key**. which is the attribute that can uniquely identify weak entities that are related to the *same owner entity*. In a worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.
    - Weak entity types can sometimes be represented as complex (composite, multivalued) attributes.

0.14 Proper Naming of Schema Constructs

- *Singular Names* are chosen to named for entity types, rather than plural names because the entity type name applies to each individual entity belonging to that entity type.
- Entity type and relationship type names are in uppercase letters, attribute names have their initial letter capitalized, and role names are in lowercase letters.

0.15 Notation for ER Diagrams

0.16 Other Notation: UML Class Diagrams

0.17 Relationship Types of Degree Higher than Two

0.18 Choosing between Binary and Ternary (or Higher-Degree) Relationships

0.19 Constraints on Ternary (or Higher-Degree) Relationships

0.20 Another Example: A UNIVERSITY Database

End

Chapter IV: The Enhanced Entity - Relationship (EER) Model

0.21 Subclasses, Superclass's, and Inheritance

"The EER model includes *all* the *modeling concepts of the ER model*. In addition, it includes the concepts of **subclass** and **superclass** and the related concepts of **specialization** and **generalization**."

1. **Subclass:** A set of subset of the entities that belong to the entity set. Subclasses can also have **specific** (or local) *attributes* and **relationships**.
2. **Superclass:** An entity set of subclasses.
3. An entity cannot exist in the database merely by being a member of a subclass, it must also be a member of the superclass, an entity can be also included as a member of any number of subclasses.
4. **Inheritance:**
    - An entity in the subclass represents the same real-world entity from the superclass, it should process values for its specific attributes *as well as* values of its attributes as a member of the superclass.

→ The subclass **inherits** all the attributes of the entity as a member of the superclass.
→ The entity also **inherits** all the relationships in which the superclass participates.

0.22 Specialization and Generalization

1. **Specialization:**
    - **Specialization:** is the process of defining *a set of subclasses* of an entity type; this entity type is called the *superclass* of specialization.
    - There are two main reasons for including class/subclass relationships and specializations:
        - The first:
            * Certain attributes may apply to some but not all entities of the superclass entity type.
            * A subclass is defined in order to group the entities to which these attributes apply.
            * The members of the subclass may still share the majority of their attributes with the other members of the superclass.
        - The second:
            * Some relationship types may be participated in only by entities that are members of the subclass.
2. **Generalization**
    - A *reverse process* of abstraction in which we suppress the differences among the several entity types, identify their common features, and **generalize** them into a single **superclass**.
    - the Generalization process can be viewed as being functionally the inverse of the specialization process.

0.23 Constraints and Characteristics of Specialization and Generalization Hierarchies

1. **Constraints on Specialization and Generalization**

   - In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such classes are called **predicate-defined** (or **condition-defined**) subclasses.
   - If *all* subclasses in a specialization have their membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization**, and the attribute is called **defining attribute** of the specialization.
   - When we do not have a condition for determining membership in a subclass, the subclass is called **user-defined**.
   - **Disjointness constraint:**
     - It specify that the subclass of the specialization must be disjoint sets.
     - This means that an entity can be a member of at *most* one of the subclasses of the specialization.
     - A specialization that is attribute-defined implies the disjointness constraints. (if the attribute used to define the membership predicate is single-valued).
     - The **d** in the circle stands for *disjoint*, the **d** notation also applies to user-defined subclasses of a specialization that must be disjoint.
   - **Overlapping constraint:**
     - the same(real-world) entity may be a member of more than one subclass of the specialization.
     - The **o** notation is used to present overlapping.
   - **Completeness** (or **totalness** constraint):
     - A **total specialization** constraint specifies that *every entity* in the superclass must be a member of at least one subclass in the specialization.
     - A **partial specialization** constraint allows an entity not to belong to any of the subclasses.
     - The disjointness and completeness constraints are *independent*:
       * Disjoint, total
       * Disjoint, partial
       * Overlapping, total
       * Overlapping, partial
     - A superclass that was identified through the *generalization* process usually is **total**, because the superclass is *derived from* the subclasses and hence only the entities that are in the subclasses.
   - For insertion and deletion rules to specialization (and generalization) as a consequence of the constraints should follow the rules:
     - Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.
     - Inserting an entity in a superclass implies that the entity is **mandatorily** inserted in all *predicate-defined* (or attribute defined) subclasses for which the entity satisfies the defining predicate.
     - Inserting an entity in a superclass of a *total specialization* implies that the entity is madatorily inserted in at least one of the subclasses of the specialization

0.24 Specialization and Generalization Hierarchies and Lattices

- A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specialization.
- A **specialization hierarchy** has the constraint that every subclass participate **as a subclass** in *only one class/subclass* relationship → Each subclass has only one parent, which **result** in a **tree structure** or **strict hierarchy**.
- For a **specialization lattice**, a subclass can be a subclass in *in more than* class/subclass relationship.
- A subclass inherits the attributes not only of its direct superclass, but also of all its predecessor subclasses *all the way to the root* of the hierarchy or lattice if necessary.
- A **leaf node** is a class that has *no subclasses of its own.*
- A subclass with *no more than one* superclass is called a **shared subclass** → This lead to the concept known as **multiple inheritance**
- The existence of at least one shared subclass leads to a lattice (and hence to *multiple inheritance*); if no shared subclasses existed, we would have a hierarchy rather than a lattice and only **single inheritance** would exist.
- If an attribute (or relationship) originating in the *same superclass* is inherited more than once via different paths in the lattice, then it should be included only once in the shared subclass.

0.25 Utilizing Specialization and Generalization in Refining Conceptual Schemas

- **top-down conceptual refinement** or **bottom-up conceptual refinement**

0.26 Modeling of UNION Types Using Categories

"A subclass represent a collection of entities that is a subset of the UNION of entities from distinct entity types, that subclass is called a **union type** or **category**"

1. A category has two or more superclass that may represent collections of entities from *distinct entity types*, whereas other superclass/subclass relationships always have a single superclass.
2. (Read textbook) Differences between a lattice and a category.
3. (Read textbook) Difference between a generalization and a category.
4. A category can be **total** or **partial**:
    - A total category holds the *union* of all entities in its superclass.
    - A partial category can hold the *subset of the union.*

End

Chapter 5: The Relational Data Model and Relational Database Constraints

- The relational model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper.
- The model uses the concept of a *mathematical relation* - which looks somewhat like a table of values.
- The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS.

0.27 Relational Model Concepts

- The relational model represents the database as a collection of *relations*.
- When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values.
  - A row represents a fact that typically corresponds to a real-world entity or relationship.
  - The table name and column names are used to help to interpret the meaning of the values in each row.
- in the formal relational model terminology, a row is called *tuple*, a column header is called an *attribute*, and the table is called a*relation*.
- The data type describing the types of values that can appear in each column is represented by a *domain* of possible values.

1. **Domains, Attributes, Tuples, and Relations**
   - A **domain** D is set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned.
   - A domain is given a name, data type, and format.
   - A **relational schema** R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes, $A_1, A_2, \ldots, A_n$.
     - Each **attribute** A_i is the name of a role played by some domain D in the relation schema R.
     - D is called the **domain** of A_i and is denoted by **dom(A_i)**.
     - A relation schema is used to *describe* a relation; R is called the **name** of this relation.
     - The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.
   - For example, A relation of degree seven, which store information about university students, would contain seven attributes describing each student as follow:

     STUDENT(Name, Ssn,`Home_phone`, Address, `Office_phone`, Age, Gpa)
   - Using data type of each attribute, the definition is sometimes written as:

     STUDENT(Name: string, Ssn: string, `Home_phone`: string, Address: string, `Office_phone`: string, Age: integer, Gpa: real)
   - We can specify the domain for STUDENT: dom(Gpa) = `Grade_point_averages`.

- A **relation** (or **relation state**) r of the relation schema R($A_1, A_2, \ldots, A_n$) also denoted by r(R), is a set of n-tuples r = $t_1, t_2, \ldots, t_m$.
- Each n-**tuple** t is an ordered list of n values t =< $v_1, v_2, \ldots, v_n$ >, where each value v_i, $1 \leq i \geq n$, is an element of dom(A_i) or is a special NULL value.
- The ith value in tuple t, which corresponds to the attribute A_i, is referred to as t[A_i] or t.A_i (or *t[i]* if we use the positional notation).
- A **relation intension** is used for schema R and the term **relation extension** is used for a relation state r(R).
- **Cardinality** is the number of *tuples* in the table.
- It is possible for several attributes to *have the same domain*. The attribute names indicate different **roles**, or interpretations, for the domain.

2. **Characteristic of Relations**

- **Ordering of Tuples in a Relation:**
  - A relation is defined as a *set of tuples*; elements of a set have *no order* among them; hence tuples in a relation do not have any particular order.
  - A relation is not sensitive to the ordering of tuples. When we display a relation as a table, the rows are displayed in a certain order.
- **Ordering of Values within Tuple and an Alternative Definition of a Relation:**
  - An **alternative definition** of a relation can be given, making the ordering of values in a tuple unnecessary. In this definition, a relation schema R = $A_1, A_2, \ldots, A_n$ is a set of attributes (instead of an ordered list of attributes), and a relation state r(R) is a finite set of mappings r = $t_1, t_2, \ldots, t_m$, where each tuple ti is a mapping from R to D, and D is the union (denoted by $\cup$) of the attribute domains; that is, $D = dom(A_1) \cup dom(A_2) \cup \ldots \cup dom(A_n)$. In this definition, $t[A_i]$ must be in $dom(A_i)$ for $1 \leq i \leq n$ for each mapping t in r. Each mapping t_i is called a tuple.
  - According to this definition of tuple as a mapping, a tuple can be considered as a set of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $dom(A_i)$.
  - The ordering of attributes is not important, because the attribute name appears with its value.
  - When the attribute name and value are included together in a tuple, it is known as **self-describing data**, because the description of each value (attribute name. is included in the tuple.
- **Values and NULLs in the Tuples:**
  - Each value in a tuple is an **atomic** value.
  - Composite and multivalued attributes are not allowed. In relational model, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes.
  - **flat relational model**, must of theory behind the relational model was developed with this assumption in mind, which is called **first normal form** assumption.
  - In general, we can have several meanings for NULL values, such as **value unknown, value** exists but is **no available**, or **attribute does not apply** to this tuple (also known as **value undefined**).
- **Interpretation (Meaning) of a Relation**
  - The relation schema can be interpreted as a declaration or a type of **assertion**.

  - Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion.
  - Some relations may represent facts about *entities*, whereas other relations may represent facts about *relationships*.
  - An alternative interpretation of a relation schema is as a **predicate**, the values in each tuple are interpreted as values that *satisfy* the predicate.

3. **Relational Model Notation**

- A relation schema R of degree n is denoted by $R(A_1, A_2, \ldots, A_n)$.
- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- The name of a relation schema indicates the current set of tuples in that relation - the *current relation state* - refers *only* to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using dot notation. For example: STUDENT.Name or STUDENT.Age
- All attribute names in a *particular relation* must be distinct.
- An n-tuple t in a relation r(R) is denoted by $t =< v_1, v_2, \ldots, v_n >$, where v_i is the value corresponding to attribute A_i.
  - Both $t[A_i]$ and $t.A_i$ (and sometimes t[i]) refer to the value v_i in t for attribute A_i.
  - Both $t[A_u, A_w, \ldots, A_z]$ and $t.(A_u, A_w, \ldots, A_z)$, where $A_u, A_w, \ldots, A_z$ is a list of attributes from R, refer to the subtuple of values $< v_u, v_w, \ldots, v_z >$ from t corresponding to the attributes specified in the list.

0.28 Relational Model Constraints and Relational Database Schemas

- Constraints on databases can be generally be divided into three main categories:
  - Constraints that are inherent in the data model → **inherent model-based constraints** or **implicit constraints**
  - Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL (data definition language) → **schema-based constraints** or **explicit constraints**.
  - Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. → **application-based** or **semantic constraints** or **business constraints**.
  - The characteristics of relations are the inherent constraints of the relational model.
  - Another important category of constraints is *data dependencies*, which include *functional dependencies* and *multivalued dependencies*.

1. **Domain Constraints**

- Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain dom(A).

2. **Key constraints and Constraints on NULL Values**

- In the formal relation model, a *relation* is defined as a set of *tuples* - All elements of a set are distinct → all tuples in a relation must be also distinct.
  - A **superkey** specifies that a *uniqueness constraints* that no more two distinct tuples in any state r of R can have the same value set of attributes.
  - Each relation has at least one default superkey - the set of all its attributes.
  - A superkey can have redundant attributes
  - A **key** k of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attribute *K'* that is not a superkey of R any more. A key satisfies two properties:
    - Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key.
    - It is a *minimal superkey* - that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraints hold. This *minimality* property is required for a key but is optional for a superkey.
  - A key with multiple attributes must require *all* its attributes together to have the uniqueness property.
  - The value of key attribute can be used to identify uniquely each tuple in the relation.
  - A set of attributes constituting a key is a property of the schema; it is constraints that should hold on every valid relation state of the schema.
  - A key is determined from the meaning of the attributes, and the property is *time-invariant.*
  - A relation schema may have more than one key, 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. → the candidate key whose values are used to *identify* tuples in the relation.
  - Attributes that form the primary key of a relation schema are underlined.
  - The other candidate keys that are not chosen as the primary key are designate as **unique keys** and are not underlined.

3. **Relational Databases and Relational Database Schemas**

   - A **relational database schema** S is set of relation schemas $S = R_1, R_2, \ldots, R_m$ and a set of **integrity constraints** IC.
   - A **relational database state** DB of S is a set of relation states $DB = r_1, r_2, \ldots, r_m$ such that each r_i is a state of R_i, and such that the r_i relation states satisfy the integrity constraints specified in IC.
   - A database state does not obey the integrity constraints is called **not valid** while a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state**.
   - Attributes that represent the same real-world concept may or may not have identical names in different relations. However we could have two attributes that share the same name but represent different real-world concepts.
   - When we represent an attribute, it would have *identical attribute* names in all relations. This create problems when the same new-world concept is used in different roles (meanings) in the same relation. → require a distinct attribute names to distinguish their meaning.
   - Each relational DBMS must have a data definition language (DDL) for defining relational database schema.
   - Integrity constraints are specified on a database schema and are expected to hold on every *valid database state* of that schema.

4. **Entity Integrity, Referential Integrity, and Foreign Keys**

- The **entity integrity constraints** states that no primary key value can be NULL. (primary key value is used to identify individual tuples in a relation).
- Key constraints and entity integrity constraints are specified on individual relations.
- The **referential integrity constraints** is specified between two relations and is used to maintain the consistency among tuples in the two relations.
- The referential integrity constraints states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation.
- To define *referential integrity* → the concept of *foreign key* (FK). A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies:
  - The attributes in FK have the same domain(s) as the primary key attributes (PK) of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.
  - A value of FK in a tuple t_1 of the current state r_1 (R_1) either occurs as a value of PK for some tuple t_2 in the current state r_2 (R_2) or is *NULL*. iN the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 **references** or **refers to** to the tuple t_2.
  - R_1 is called the **referencing relation** and R_2 is the **referenced relation**.
  - A foreign key can *refer to its own relation.*
  - We can diagrammatically display referential integrity constraints by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation.

5. **Other Types of Constraints:** (read textbooks if necessary)

0.29 Update Operations, Transactions, and Dealing with Constraints Violations

- The operations of a relational model can be categorized into *retrievals* and *updates.*
- The relational algebra operations can be used to specify **retrievals**.
- A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information.
- The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. The **result relation** becomes the answer to the user's query.
- **relational calculus** is used to define a query declaratively without giving a specific order of operations.
- There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify):
  - **Insert** is used to insert one or more new tuples in a relation.
  - **Delete** is used to delete tuples.
  - **Update** (or **Modify**) is used to change the values of some attributes in existing tuples.
- Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated.

1. **The Insert Operation**
   - The **Insert** operation provide a list of attribute values for new tuple t that is to be inserted into a relation R. Insert can violate any of the 4 types of constraints:

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

2. **The Delete Operation**

- The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database.
- If a deletion causes a violation, there are 2 options:
  - **restrict:** *reject the deletion.*
  - **cascade:** *deletes tuples that reference the tuple that is being deleted.* Note that if a referencing attribute that causes a violation is *part of the primary key*, it *cannot* be set to NULL; otherwise it would violates the entity integrity.
  - In general, when a referential integrity constraints is specified in the DDL, the DBMS will allow the database designer to *specify which of the options* applies in case of a violation of the constraints.

3. **The Update Operation**

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

0.30 The Transaction Concept

- A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database.
- At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema.
- A single transaction may involve any number of retrieval operations and any number of update operations. These retrieval and updates will together form an atomic unit of work against the database.

End

Chapter 6: Basic SQL

- The name SQL is presently expanded as **Structural Query Language**. (or SEQUEL: Structured English QUEry Language)
- SQL was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R.
- SQL is now the standard language for commercial relational DBMSs.
- SQL is a comprehensive database language; It has statements for data definitions, queries, and updates. (DDL and DML).
- It has facilities for defining views on the database, specifying security and authorization for defining integrity constraints, and specifying transaction controls.
- It has rules for embedding SQL statement into a general-purpose programming language such as Java/C++.

0.31 SQL Data Definition and Data Types

- SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*.
- The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables, types, and domains as well as views, assertion, and triggers.

1. **Schema and Catalog Concepts in SQL**
   - An **SQL Schema** is identified by a **schema name** and includes an **authorization identifier** to indicate the user or a account who owns the schema, as well as **descriptor** for *each element* in the schema.
   - **Schema elements** include tables, types, constraints, views, domains and other constructs (such as authorization grants) that describe schema.
   - A schema is created via the CREATE SCHEMA statement, which can include all the schema element's definition, or the schema can be assigned a name and authorization identifier and the elements can be defined later.
   - Each statement in SQL ends with a semicolon (';').
   - The following statement called COMPANY owned by the user with authorized identifier 'Jsmith'.

     CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmith';
   - The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or a DBA (DataBase Administrator)
   - SQl uses concept **catalog** - a collection of schemas.
   - A catalog always contains a special schema $INFORMATION_SCHEMA$, which provides information on all the schemas in the catalog and all the element descriptor in these schemas.
   - Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog.

- Schemas within the same catalog can also share certain elements, such as type and domain definitions.

2. **The CREATE TABLE Command in SQL**

   - The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints.
   - The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and possibly attribute constraints, such as NOT NULL.
   - The key, entity integrity, and referential integrity constraints can be specified within CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command;
   - For instance, to create a EMPLOYEE relation explicitly:

     CREATE TABLE EMPLOYEE

   - Or, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Example:

     CREATE TABLE COMPANY.EMPLOYEE

   - The relations declared through CREATE TABLE statements are called **base table** (or base relations); this means that the table and its rows are actually created.

Relational Database Design by ER/EER-to-Relational Mapping

1. Mapping of Regular Entity Types:
   - For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E.
   - Include only the simple component attributes of a **composite** attribute.
   - Choose one of the key attributes of E as the primary key for R. If the chosen key of E is a composite, then the **set** of simple attribute that form it will together form the primary key of R.
   - If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify additional (unique) keys of relation R.
2. Mapping of Weak Entity Types:
   - For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R.
   - The primary key attribute(s) of the relation(s) that correspond to the owner entity type(s) are included as foreign key of R, this **takes** care of mapping the idenifying relationship type of W.
   - The primary key of R is the **combination** of the primary key(s) of the owner(s) and the partial key of the weak entity type W.
   - If there is a weak entity type E2, whose owner is also a weak entity type E1, then E1 should be mapped first before E2 to determine its primary key first.
3. Mapping of Binary 1:1 Relationship Types:
   - For each binary 1:1 relationship type R in the ER schema, identify the relation S and T that correspond to the entity types participating in R.
   - There are 3 possible approaches to illustrate it:
     - Foreign key approach:
       * Choose one relation S and include the primary key of T as the foreign key in S.
       * It is important to choose an entity type with *total participation* in R in the role of S.
       * Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.
       * It is possible to include the primary key of S as a forein key in T instead. However, it will have **NULL** value for T tuples that does not participating in R relationship type with S $\rightarrow$ Create redundancy and incurs a penalty for consistency maintenance.
     - Merged relation approach:
       * An alternating mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation.
       * This is possible when *both participation* are *total*, as this would indicate that the two tables have the same number of tuples at all times.
     - Cross-reference or relationship relation approach:
       * Set up the third relation R for the purpose of cross-referencing the primary keys of the two relations S and T reprensenting the entity types.
       * The third relation R is called **a relationship relation** (or sometimes called a **lookup table**).

    - The third relation R includes the primary key attributes of S and T as foreign keys to S and T. The primary key of R will be one of the two foreign keys, and the other will be a unique key of R.
    - One drawback is having an extra relation, and requiring extra join operations when combining related tuples from the tables.

4. Mapping of Binary 1:N Relationship Types: 2 approaches for mapping this type of relationship
    - The foreign key approach
        - For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the *N*-side of the relationship type.
        - Include the primary key of T as the foreign key of S.
        - Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type R as attributes of S.
    - The relationship relation approach
        - We create a separate relation R whose attributes are the primary keys of S and T, which will also be the foreign keys to S and T. The primary key of R is the same as the primary key of S.

5. Mapping of Binary M:N Relationship Types:
    - To illustrate this mapping type we use the **relationship relation** (cross-reference) option.
    - For each binary M:N relationship type R, create a new relation S to represent R.
    - Include the primary keys of the relations that represent the participating entity types as foreign key attributes in S.
    - The *combination* of the primary keys that represent the participating entity types will form the primary key of S.
    - Include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S.

6. Mapping of Multivalued Attributes
    - For each multivalued attribute A, create a relation R.
    - R will include an attribute correspond to A, plus the primary key attribute K of the relation that represents the entity type or relationship type that has A as a multivalued attribute - as foreign key of R.
    - If the multivalued attribute A is composite we will include its simple components.
    - The primary key of R is the combination of simple attributes in A and the primary key attribute K.
    - In some cases, when a multivalued attribute is composite, only some of the component attributes are required to be part of the key of R.

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

- The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types.
- If the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E' corresponding to E.

Mapping EER Model Constructs to Relations

1. Mapping of Specialization or Generalization:
    - Options for Mapping Specialization or Generalization:
        - Convert each specialization with m subclasses $S1, S2, \ldots, Sn$ and (generalized) superclass C, where attribtutes of C are $k, a1, a2, \ldots, an$ and k is the (primary) key, into relation schemas using one of the following options:
            * Multiple relations - superclasses and subclasses:
                · Create a relation for C with all the attribute in C and the primary key is k.
                · Create relation for each subclass with the attribute is k + attribute of each subclass where k is the primary key.
                · This option works for any specialization (total or partial, disjoint or overlapping).
            * Multiple relations - subclass relations only:
                · Create a relation for each subclass with the attributes are the attributes of each subclass and the attributes of superclass C where k is the primary key.
                · This option works only for a specialization whose subclasses are *total* (every entity in the superclass must belong to (at least) one of the subclass).
                · Additionally, it is recommended if the specialization has the *disjointness constraints*, else if the specialization is overlapping, the same entity may be duplicated in several relations.
            * Single relation with one type attribute:
                · Create a single relation with attributes of superclass C and *every attributes* of each subclass and attribute t called **type** indicates the subclass to which each tuple belongs to , k is primary key of the relation.
                · This option works only for a specialization whose subclasses are *disjoint*, the drawback is that this relations can generate many NULL values is many specific attributes exist in the subclass.
            * Single relation with multiple type attributes:
                · Create a single relation schema L with attributes of each subclass and subperclass with each flags indicate the following subclass.
                · This option works for a specialization whose subclasses are *overlapping*