Chapter Two

Relational Model

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package is one tightly integrated system, to the modern DBMS packages that are modular in design, with client-server system architecture. This evolution mirrors the trends in computing, where the large centralized mainframe computers are being replaced by hundreds of distributed workstations and personal computers connected via communications networks.

In basic client-server DBMS architecture, the system functionality is distributed between two types of modules. A ***client module*** is typically designed so that it will run on a user workstation or personal computer. Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as forms or menu-based GUIs (graphical user interfaces). The other kind of module, called a ***server module***, typically handles data storage, access, search, and other functions.

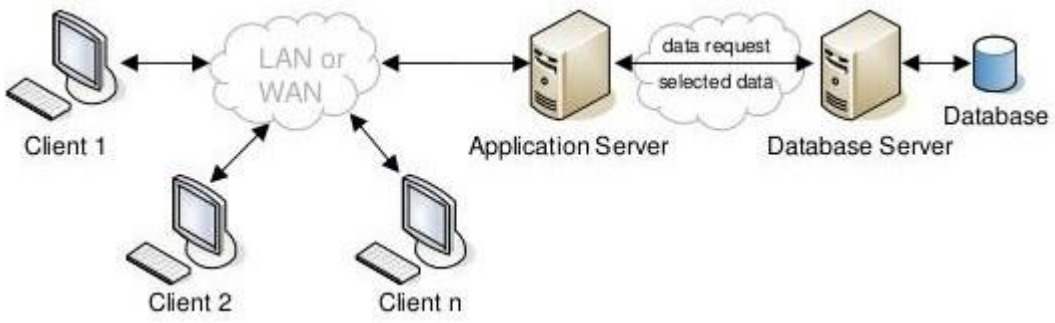

Figure 2.1: Three-Tier Client-Server Architecture

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 not needed by most database users. A **data model**: a collection of concepts that can be used to describe the structure of a database provides the necessary means to achieve this abstraction. By *structure of a database* we mean the data types, relationships, and constraints that should hold on the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database. In addition to the

basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the ***dynamic aspect or behavior*** of a database application. This allows the database designer to specify a set of valid ***user-defined*** operations that are allowed on the database objects. An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the basic data model operations.

2.1.1. Categories of Data Models

Many data models have been proposed, and we can categorize them according to the types of concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an interaction among the entities.

2.1.2. Schema and Instances

In any data model it is important to distinguish between the *description* of the database and the *database itself.* The description of a database is called the **database schema (see figure 2.3),** which is specified during database design and is not expected to change frequently. The actual data in a database may change quite frequently; for example, the database shown in Figure 2.3 changes every time we add an employee or enter a new salary for an employee. The data in the database at a particular moment in time is called a **database state** or **snapshot.** It is also called the *current* set of **occurrences** or **instances** in the database.

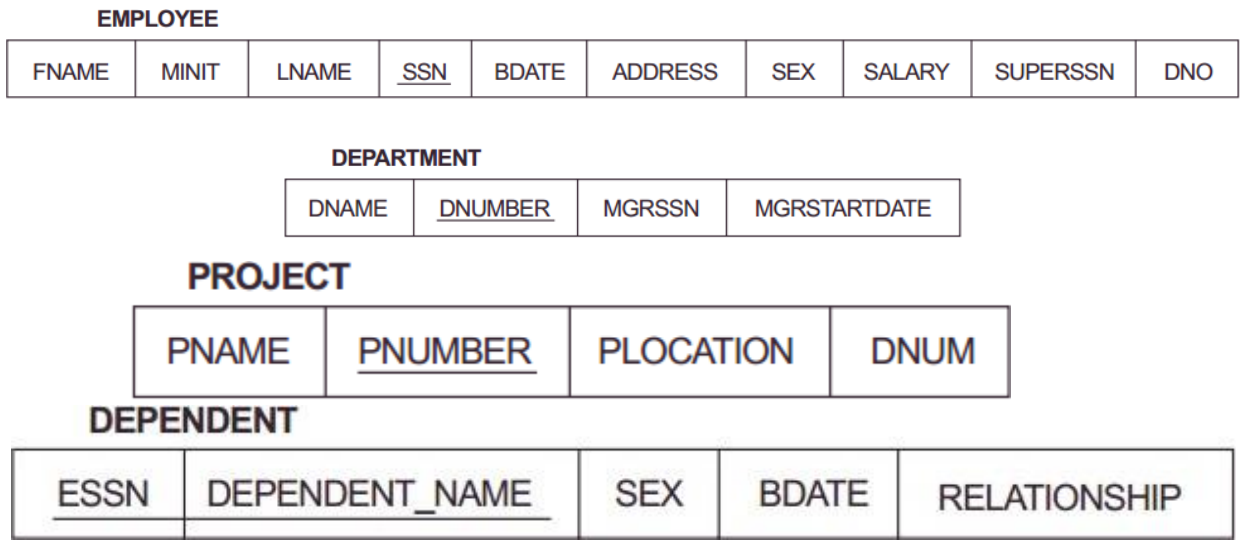

Figure 2.3: Schema diagram for company

2.2. Three Schema Architecture and Data Independence

In this section we specify architecture for database systems, called the **three-schema architecture**, which was proposed to help achieve and visualize characteristics of database approach.

2.2.1. The Three-Schema Architecture

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

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is

interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

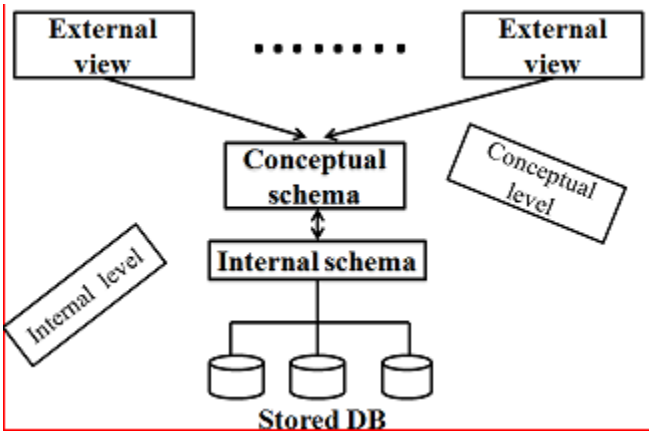

Figure 2.2: Three-Schema Architecture

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

2.2.2 Data Independence

The three-schema architecture can be used to explain the concept of **data independence,** which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must

work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

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

2.3. Entity, Attribute and Relationship

1. The **ENTITIES** (persons, places, things etc.) which the organization has to deal with.

Relations can also describe relationships

- ✓ The name given to an entity should always be a singular noun descriptive of each item to be stored in it. E.g.: student NOT students.
- ✓ Every relation has a schema, which describes the columns, or fields
- ✓ The relation itself corresponds to our familiar notion of a table: A relation is a collection of tuples, each of which contains values for a fixed number of attributes

Types of Entity

- ✓ **Strong Entity:** An entity whose existence does not depend on the existence of another entity.
  : An entity that can have a primary key
  E.g. PATIENT, EMPLOYEE
- ✓ **Weak Entity:** Is an entity that can't exist in the database unless instances of one or more other entities exist in the database. Entity types that do not have key attributes of their own.
  E.g. PRESCRIPTION, CHILDREN OF EMPLOYEE

2. The **ATTRIBUTES** - the items of information which characterize and describe these entities.

Attributes are pieces of information ABOUT entities. The analysis must of course identify those which are actually relevant to the proposed application. Attributes will give rise to recorded items of data in the database

At this level we need to know such things as:

• Attribute name (be explanatory words or phrases)

• Whether the attribute is part of the entity identifier (attributes which just describe an entity and those which help to identify it uniquely)

• Whether it is permanent or time-varying (which attributes may change their values over time)

• Whether it is required or optional for the entity (whose values will sometimes be unknown or irrelevant)

Types of Attributes

(1) Simple (atomic) Vs Composite attributes

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, (***Address, Name***) the Address attribute can be sub-divided into zone, wereda, kebele. Attributes that are not divisible are called **simple** or **atomic attributes. Simple:** contains a single value (not divided into sub parts), E.g. ***Age, gender***

(2) Single-valued Vs multi-valued attributes

Most attributes have a single value for a particular entity; such attributes are called single-valued. For example, (***sex, ID, age***) Age is a single-valued attribute of person. In some cases an attribute can have a set of values for the same entity—for example, (***Telephone, color, college degree***) Colors attribute for a car, or a College Degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two values for Colors. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; so different persons can have different numbers of values for the College Degrees attribute. Such attributes are called multivalued.

(3) Stored vs. Derived Attribute

In some cases two (or more) attribute values are related—for example, the ***Age*** and ***Birth Date*** attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth Date. The Age attribute is hence called a derived attribute and is said to be derivable from the Birth Date attribute, which is called a stored attribute. Some attribute values can be derived from related entities; for example, an attribute Number of Employees of a department entity can be derived by counting the number of employees related to (working for) that department.

(4) Null Values

• **NULL** applies to attributes which are not applicable or which do not have values.

• You may enter the value NA (meaning not applicable)

• Value of a key attribute cannot be null.

Default value - assumed value if no explicit value

3. The **RELATIONSHIPS**:-is an association between entities so that our database represents the real world.

- ✓ It is represented by lines between entities.
- ✓ A relationship should be named by a word or phrase which explains its function (**role name**)
- ✓ Each entity type that participates in a relationship type plays a particular role in the relationship

DEGREE OF RELATIONSHIPS

- ✓ Indicates the number of entities involved in a relationship
- ✓ Based on the number of participating entities in a relationship we have different types of relationships: *Unary/* Recursive, *Binary, and Ternary*

Unary

- ✓ Number of entities involved in the relationship is only one
- ✓ The relationship is b\n different instances of the same entity
- ✓ The same entity participates in different roles

E.g. *EMPLOYEE* and *MANAGER* relationship, *PREREQUISITE* & *COURSE*

Binary

- ✓ The number of participating entities is two
- ✓ The relationship is b\n instances of different entities
- ✓ It is the most common type of relationship

E.g. *STUDENT* and *DORMITORY* || *STUDENT* and *COURSE*

Ternary

- ✓ The number of entities involved are three

E.g. *INSTRUCTOR* teaches *STUDENTS* of a department a particular *COURSE*

Cardinality Ratios for Binary Relationships

The cardinality ratio for a binary relationship specifies the number of relationship instances that an entity can participate in.

ONE-TO-ONE, each row in one database table is linked to 1 and only 1 other row in another table. E.g. of a 1:1 binary relationship is EMPLOYEE *MANAGES* DEPARTMENT (*see ER-diagram for company in chapter 3*). This means any point in time an employee can manage only one department and a department has only one manager.

ONE-TO-MANY, one tuple can be associated with many other tuples, but not the reverse.

E.g. *DEPARTMENT-EMPLOYEE (see ER-diagram for company in chapter 3)*: as one department can have multiple employees, in the WORKS_FOR binary relationship type, the cardinality ratio is1: N, meaning that each department can be related to any number of employees, but an employee can work for only one department.

MANY-TO-MANY, one tuple is associated with many other tuples and from the other side, with a different role name one tuple will be associated with many tuples

E.g. *STUDENT – COURSE* as a student can take many courses and a single course can be attended by many students.

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, and is sometimes called the minimum cardinality constraint. There are two types of participation constraints:
  - ❖ **total and partial**
- ✓ If a company policy states that every employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_ FOR relationship instance .Thus, the participation of EMPLOYEE in WORKS_FOR is called total participation, meaning that every entity in "the total set" of employee entities must be related to a department entity via WORKS_FOR.
- ✓ Total participation is also called existence dependency.
- ✓ We do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is partial, meaning that some or "part of the set of" employee entities are related to some department entity via MANAGES, but not necessarily all.

- ✓ We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

2.4. Relational Constraints/Integrity Rules and keys

- ✓ Each row of a table is uniquely identified by a *PRIMARY KEY* composed of one or more columns
- ✓ Each tuple in a relation must be unique
- ✓ Group of columns, that uniquely identifies a row in a table is called a *CANDIDATE KEY*
- ✓ A candidate key has two properties:
    1. **Uniqueness**
    2. **Irreducibility**
- ✓ A column or combination of columns that matches the primary key of another table is called a *FOREIGN KEY* which is used to cross-reference tables.
- ✓ *ENTITY INTEGRITY RULE* of the model states that no component of the primary key may contain a NULL value.
- ✓ The *REFERENTIAL INTEGRITY RULE* of the model states that, for every foreign key value in a table there must be a corresponding primary key value in another table in the database or it should be NULL.
- ✓ DOMAIN INTEGRITY**:** No value of the attribute should be beyond the allowable limits
- ✓ ENTERPRISE INTEGRITY**:** Additional rules specified by the users or database administrators of a database are incorporated

2.3.2. Building Blocks of the Relational Data Model

The components of the relational data model are:

- ❖ **Entities:** real world physical or logical object
- ❖ **Attributes:** properties used to describe each Entity or real world object.
- ❖ **Relationship:** the association between Entities
- ❖ **Domain:** is a set of values
- ❖ **Constraints:** rules that should be obeyed while manipulating the data.
- ❖ **Schema:** framework of the database