CSCI3170 Introduction to Database Systems

Ryan Chan

October 13, 2025

Abstract

This is a note for **CSCI3170 Introduction to Database Systems**.
Contents are adapted from the lecture notes of CSCI3170, prepared by Michael Ruisi Yu, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

Chapter 1

Introduction

1.1 Overview

Data will always need to be stored, manipulated, accessed, shared, and transmitted. Thus, we require certain methods to handle it.

A data table (or data frame) is a two-dimensional structure. Regarding the data itself, we generally categorize it into three types. The first is **Unstructured Data**, which refers to information that does not follow a specific format, such as the statement: "a university has 10,000 students." Next, we have **Semi-structured Data**, which has some organizational elements (e.g., tags, hierarchies) but is still difficult to process directly. Finally, we have **Structured Data**, the most organized type, stored in predefined formats such as tables with rows and columns.

To manage such vast amounts of data, we use **Database Management Systems (DBMS)**, which are software packages designed to maintain and utilize large collections of data.

By using a DBMS to store data, we ensure data independence, data integrity, security, concurrent access, and crash recovery.

First, we begin with the **conceptual model**.

1.2 Entity Relationship Model

By receiving a set of requirements, we need to design an application. For the data behind it, we must create a database schema that provides a logical view of the data model. However, the main challenge is how to build a database for this application that meets all the requirements — in other words, how to construct a systematic database.

In this case, we use Chen's **Entity-Relationship (ER) Model**. In this model, there are two major components: **Entities** and **Relationships**. An **Entity** is a collection of attributes that describe an object of interest, while a **Relationship** represents the association between entities (objects). There is also a third component, the **Attribute**, which is a data item describing a property of interest.

Let us now look at the details.

1.2.1 Entity

An **Entity** is something that is distinguishable from other objects. For example, entities can be *Classrooms*, *Students*, etc. Entities represent things in the real world, and **Attributes** describe their properties.

Some attributes are simple (also called *atomic*), meaning they cannot be split into simpler components. Each simple attribute of an entity type holds one value. It is associated with a **value set** (or *domain*), which specifies the possible values that may be assigned to that attribute for each individual entity.

An entity is described using a set of attributes whose values distinguish one entity from another of the same type. An **Entity Set** is a collection of such entities (instances of the entity type).

For each attribute associated with an entity set, we must identify a domain of possible values.

To describe data in terms of a data model, we use a **schema**. For example, for a *Student* entity set, we may define the schema:

Students(sid: string, name: string, login: string, age: integer, gpa: real)

A good entity schema should include attributes that are meaningful, well-defined, and capable of being filled with valid values.

1.2.2 E-R Diagrams

Data modeling is typically divided into three tiers:

1. **Semantic (High-level or Conceptual)** — provides an initial description of the data in the enterprise.

2. **Implementation (or Record-based)** — describes how data can be organized using models such as the Relational, Network, or Hierarchical model.

3. **Low-level (or Physical)** — specifies details such as record formats, access paths, and storage structures.

To develop a database, we must first analyze the requirements, then design and implement the data model. The **Entity-Relationship (E-R) diagram** is often used for conceptual modeling.

For example, in Figure 1.1, the E-R diagram represents the following description:

> "There is one strong entity type called *Car*. It has a multivalued attribute to describe its color. It also has a composite attribute *Registration*, which consists of *Registration Number* and *State*. Additionally, it includes several other attributes such as *Make*, *Model*, and *Year*."

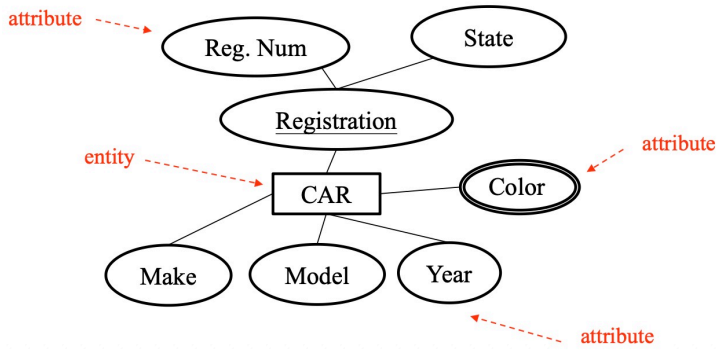


Figure 1.1: E-R Diagram: Car entity

1.2.3 Relationship

We have discussed **Entities** and **Attributes** in the Entity–Relationship (E-R) Model. The second major component is the **Relationship**.

A relationship can also have **descriptive attributes** (see Figure 1.2). Descriptive attributes are used to record information about the relationship itself, rather than about any of the participating entities.

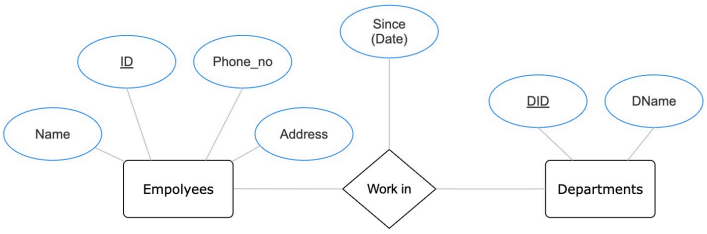


Figure 1.2: E-R Diagram: Relationship

Relationships represent logical links between two or more entities. Any set of entities is not limited to participating in only one relationship with each other.

An **Entity–Occurrence Diagram** shows the relationships between individual occurrences (instances) of particular entities.

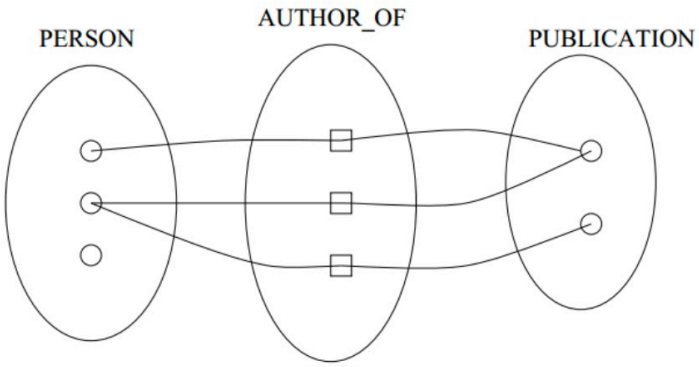


Figure 1.3: Many-to-Many Relationship

There are three main types of relationships. To distinguish among them, we apply two types of constraints: **cardinality** and **participation**.

1.2.4 Cardinality

The **cardinality constraint** (or *cardinality ratio*) specifies the number of relationship instances an entity can participate in. The three common types are:

- **1:1 (one-to-one)** — each entity instance is associated with at most one instance of another entity.
- **1:N (one-to-many)** — one entity instance can be associated with many instances of another entity.
- **M:N (many-to-many)** — multiple instances of one entity can be associated with multiple instances of another entity.

We can use standard **entity–occurrence diagrams** to visualize such relationships.

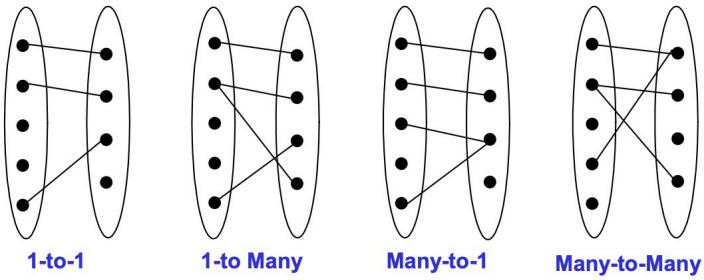


Figure 1.4: Standard Entity-Occurrence Diagrams

One-to-Many

One-to-many constraint from A to B: an entity in B can be associated with at most one entity in A.

To represent such relationships, we use arrows in the diagram. An arrow indicates that one entity can be uniquely associated with a relationship instance.

For example, in the following case, each *Employee* belongs to one *Department*, while a *Department* may have multiple employees. We use an arrow pointing from the *Employee* entity set to the *works_in* relationship to represent this constraint.

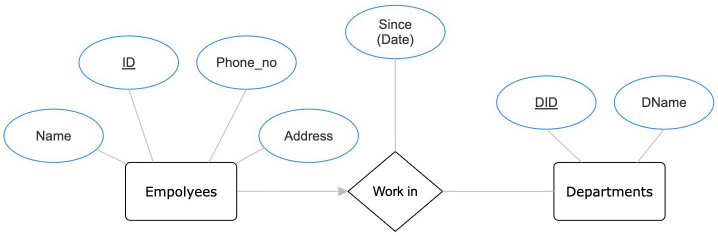


Figure 1.5: E-R Diagram: One-to-many

One-to-One

If a relationship between A and B satisfies the one-to-one mapping constraint, i.e., each entity in A is related to at most one entity in B, and each entity in B is related to at most one entity in A,

we represent this relationship with two arrows, one pointing from A to B and one from B to A.

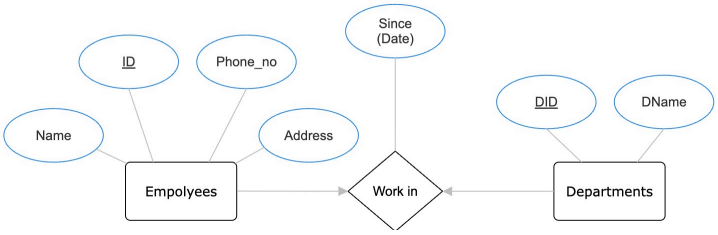


Figure 1.6: E-R Diagram: One-to-One

Many-to-Many

If an entity in A can be associated with any number of entities in B, and an entity in B can be associated with any number of entities in A, this indicates that there is no restriction on the mapping.

1.2.5 Participation

In a one-to-one relationship, as shown in Figure 1.6, if we change the *works_in* relationship to *manages*, one question arises: is there at least one manager in each department? The cardinality constraint does not provide this information. Therefore, we need **participation constraints**.

We can classify participation in relationships as follows:

- **Total participation** — every entity in the entity set must participate in at least one relationship.
- **Partial participation** — an entity in the entity set may not participate in any relationship.

In an E-R diagram, total participation is represented by a double line, while partial participation is represented by a single line.

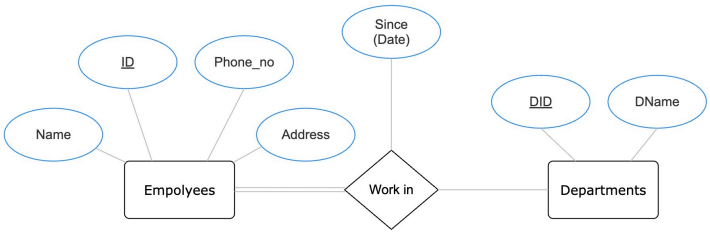


Figure 1.7: E-R Diagram: Total Participation

As shown in Figure 1.7, this indicates that every *Employee* must work for some *Department*.

1.2.6 Attributes

The final component of Entity–Relationship (E-R) modeling is **Attributes**.

Some semantics cannot be captured using simple (atomic) attributes. In such cases, we use **multivalued attributes**, which are represented by a double ellipse in an E-R diagram.

We also have **composite attributes**. Unlike simple attributes, which are indivisible, composite attributes can be divided into smaller subparts, each representing a more basic attribute with independent meaning.

Another type is **derived attributes**. These are attributes whose values can be derived from other attributes, rather than being stored directly. For example, a person's *age* can be derived from their *date of birth*. In an E-R diagram, derived attributes are represented by a dashed ellipse.

1.3 Key

In the discussion of entities, we mentioned that an entity represents a set of objects with the same attributes within a data model. How, then, can we distinguish different entities? The answer is through **keys**.

1.3.1 Overview

A key may contain more than one attribute. In some cases, we need to add attribute(s) as a key. In an E-R diagram, keys are represented by *underlined attributes*.

A **natural key** is a column or a set of columns that already exist in a table and uniquely identify a record in that table.

A **super-key** is any set of attributes that can uniquely identify an entity. If a key consists of more than one attribute, it is called a **composite key** or **compound key**.

A **candidate key** is a minimal set of attributes whose values uniquely identify an entity in the entity set. As there could be more than one candidate key, **primary key** is a candidate key chosen to serve as the main key for the entity set.

For example, consider the following schema:

Lecture(lecturer: string, course_code: string, location: string, date: date, time: string)

In this schema, {location, date, time} is a key, while {lecturer, location, date, time} is not minimal, but it is a **super-key**.

However, it is possible that no existing attribute can uniquely identify an entity. In such cases, we can provide a **surrogate key**, which is a system-generated value used to uniquely identify a record in a table.

All entity sets have a **primary key**. They are considered independent if they possess a primary key.

For entity sets that do not have a primary key, i.e., they are dependent on another entity set, they are called **weak entity sets**. In an E-R diagram, a weak entity is represented by a double rectangle.

We can make a weak entity stronger by providing a **surrogate key**. However, its existence still depends on the existence of an **identifying entity set**. A weak entity must be related to the identifying entity set via a total one-to-many relationship from the identifying entity set to the weak entity set.

A weak entity also has a **partial (discriminator) key**, which uniquely identifies weak entities only within the context of the identifying entity.

Note that in an E-R diagram, the partial key of a weak entity set is denoted with a *dashed underline*, and the identifying relationship is represented by a *double diamond*.

1.3.2 Keys in Relationships

A relationship must be uniquely identifiable. The participating entity instances contribute to the identification of each relationship instance.

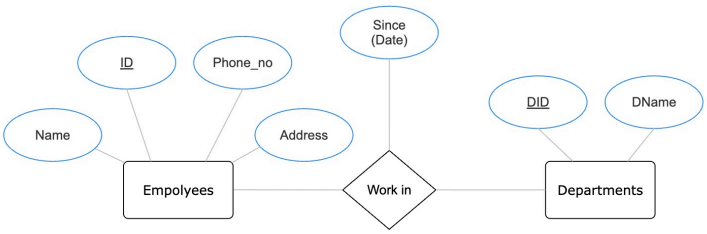


Figure 1.8: E-R Diagram

In Figure 1.8, each relationship is uniquely identified by the combination of the *Employee* ID and the *Department* DID. That is, for each pair, there cannot be more than one corresponding relationship instance.

The concept of keys is also used to identify relationships, similar to entities.

- For a relationship among $E_1, \ldots, E_k$ with no mapping constraint (many-to-many), the primary key is normally the union of the primary keys of $E_1, \ldots, E_k$.
- For a one-to-many relationship, an entity set E has a key constraint in a relationship set R, such that each entity in E participates in at most one relationship in R. Hence, an entity in E can uniquely identify a relationship in R, and the key of E can be used as the key in R.
- For a one-to-one relationship between two entity sets E and F, both key(E) and key(F) can serve as keys for the relationship set.

1.4 Conceptual Design

We say a relationship is **binary** if it links two entities.

There are also relationships of higher degree. A **ternary relationship** has degree 3. Sometimes a relationship might involve two entity instances from the same entity type; this is called a **recursive relationship**. For example, an *Employee* can be the supervisor of another employee, where this relationship links two entities within the *Employees* entity set.

For a **non-binary relationship**, suppose $n \geq 2$ for $E_1, E_2, \ldots, E_n$. It cannot always be replaced by multiple binary relationship sets. Consider the following example: the existence of (s, p), (j, p), and (s, j) does not imply the existence of (s, j, p). This phenomenon is known as the **connection trap**.

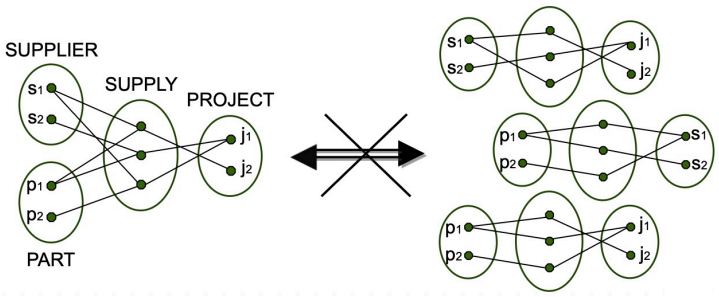


Figure 1.9: Connection Trap

In the conceptual design phase, we need to consider whether a concept should be modeled as an **attribute** or an **entity**, and whether it should be modeled as an **entity** or a **relationship**. We also need to decide whether to use **binary** or **ternary relationships**.

Sometimes, it is necessary to model a relationship between a collection of entities and relationships. **Aggregation** allows us to indicate that a relationship set (represented by a dashed box) participates in another relationship set.

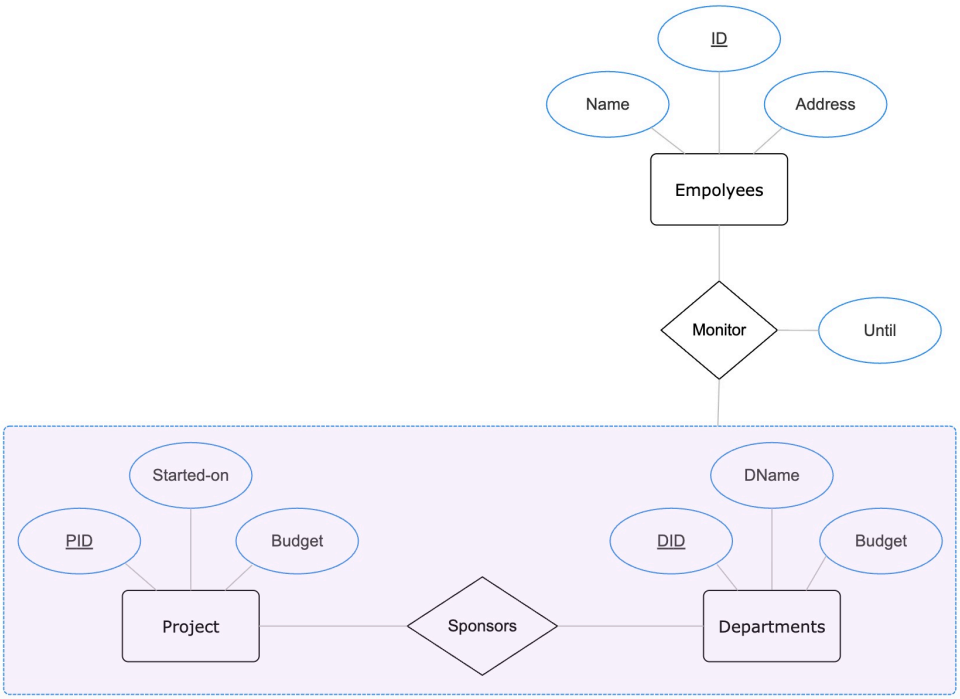


Figure 1.10: Aggregation

For example, in Figure 1.10, *Monitor* should be modeled as a relationship set that associates the *Sponsors*

relationship (rather than the *Project* or *Departments* entities) with the *Employees* entity.

1.5 Class Hierarchies

It is natural to classify the entities in an entity set into **subclasses**, because we may want to add descriptive attributes that are meaningful only for entities in a subclass, and we may want to identify which entities participate in certain relationships.

A **class hierarchy** can be viewed in two ways:

- A class is **specialized** into subclasses.
- Subclasses are **generalized** by a super-class.

For example, in *Employees*, there could be *Hourly Employees* and *Contract Employees*. Note that attributes of the super-class are inherited by entities in the subclass.

We can specify two kinds of constraints with respect to ISA hierarchies:

1. **Overlap constraints**: Determine whether subclasses are allowed to contain the same entity.

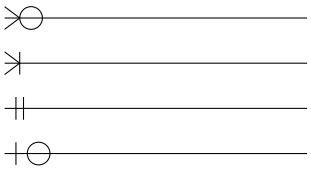
Example: Can an employee belong to both *Hourly Employees* and *Contract Employees*?

2. **Covering constraints**: Determine whether the entities in the subclasses collectively include all entities in the super-class.

Example: Does every *Employee* entity also have to be either an *hourly employee* or a *contract employee* entity?

1.6 Crows Feet Notation

1.6.1 Relationships



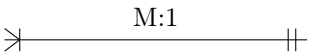
Zero or More

One or More

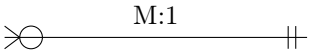
One and only One

Zero or One

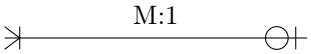
1.6.2 Many-to-One



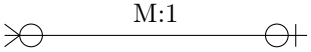
a one through many notation on one side of a relationship and a one and only one on the other



a zero through many notation on one side of a relationship and a one and only one on the other

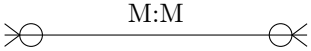


a one through many notation on one side of a relationship and a zero or one notation on the other

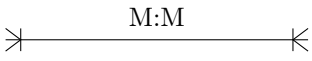


a zero through many notation on one side of a relationship and a zero or one notation on the other

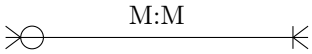
1.6.3 Many-to-Many



a zero through many on both sides of a relationship

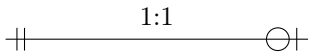


a one through many on both sides of a relationship

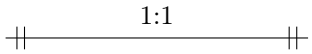


a zero through many on one side and a one through many on the other

1.6.4 One-to-One



a one and only one notation on one side of a relationship and a zero or one on the other



a one and only one notation on both sides

Chapter 2

Relation Model and Algebra

We have discussed the E-R diagram, which represents the **conceptual schema**. Once the E-R diagram is constructed, we can derive the **relational schema** (or relational diagram) from it. After examining the conversion process, we will then explore the fundamentals of **relational algebra**.

Let us first take a look at the **relational model**.

2.1 Relational Model

A **relation** consists of a **relation schema** and its **relation instance**. One can think of a relation as an *entity set*, where the relation instance is the set of records that populate the relation and corresponds to the entities in that set.

A **relation schema** describes the column headers for the table. It specifies the relation's name, the name of each field, and, if detailed, the domain of each field.

More formally, a relation schema R is a finite set of attribute names:

$$R = (A_1, A_2, \ldots, A_n)$$

For each attribute name A_i, there is a corresponding **domain** D_i, which is the set of values that A_i can take. Attribute values are required to be **atomic** (indivisible). The value `NULL` is also considered a member of every domain.

A relation r on the relation schema R is a subset of $D = D_1 \times D_2 \times \cdots \times D_n$. Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$. The current values (instance) of a relation are specified by a table. An element t of r is called a **tuple** and is represented by a row in the table.

The relational model requires that no two rows be identical. Thus, a relation is defined as a set of unique tuples (rows). This implies that the order in which the rows are listed, as well as the order of the fields, is not important.

> **Remark.** An unordered collection of elements is called a *set*, while an ordered collection of elements is called a *list*.

In the relational model, the concept of **keys** remains fundamental.

In the relational model, relationships are represented using **foreign keys**. A foreign key is an attribute (or a set of attributes) in one relation that refers to the primary key of another relation.

Formally, a set of attributes FK in a relation schema R_1 is a foreign key if these attributes have the same domains as the attributes of the primary key in another relation schema R_2, and for each tuple $t_1 \in R_1$, the value of FK either matches the value of the primary key in some tuple $t_2 \in R_2$, or is `NULL`.

2.2 Mapping Process

After introducing the terminology, we now examine how to derive a relational schema from an E-R diagram.

Step 1: Strong Entities

For each strong (or regular) entity type E, create a new relation R.

- The attributes of R include all simple attributes (and simple components of composite attributes) of E.
- The primary key of R is the key of E.

Step 2: Weak Entities

For each weak entity type W with the identifying (owner) entity type E:

- Create a new relation R.
- The attributes of R include all simple attributes (and simple components of composite attributes) of W, as well as the primary key attributes of the relation derived from E.
- The key of R is the combination of the foreign key to E and the partial key of W.

Step 3: One-to-One Relationships

For each one-to-one relationship type B, let S and T be the participating entity types. Choose one of them (say S) — preferably the one with total participation.

- Add the primary key attributes of T to the relation corresponding to S as a foreign key.
- Add all simple attributes (and simple components of composite attributes) of B to S.

Step 4: One-to-Many Relationships

For each one-to-many relationship type B, let S and T be the participating entity types, where S is on the "one" side and T on the "many" side.

- Add to the relation corresponding to T the primary key attributes of S as a foreign key.
- Add any simple attributes (or simple components of composite attributes) from the relationship B.

Step 5: Many-to-Many Relationships

For each many-to-many relationship type B, let S and T be the participating entity types.

- Create a new relation R.
- The attributes of R include the primary key attributes of S and T (as foreign keys), and all simple attributes (and simple components of composite attributes) of B.
- The primary key of R is the combination of the keys of S and T.

Step 6: Multivalued Attributes

For each multivalued attribute A of an entity E:

- Create a new relation R.
- If A is a multivalued simple attribute, the attributes of R are A and the primary key of E (as a foreign key).
- If A is a multivalued composite attribute, the attributes of R are all simple components of A and the primary key of E (as a foreign key).
- The primary key of R is the combination of all attributes in R.

Step 7: N-ary Relationships

For each n-ary relationship type ($n > 2$):

- Create a new relation R using the same approach as for many-to-many relationships.
- However, if one of the participating entity types has a participation ratio of 1, its key may serve as the primary key for R.

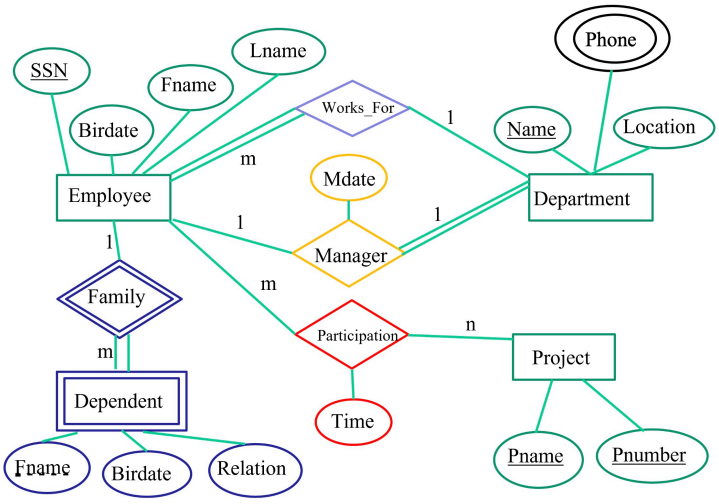


Figure 2.1: E-R Diagram

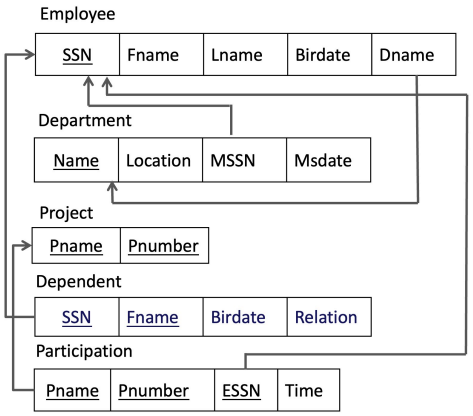


Figure 2.2: Relation Schema

2.3 Relational Integrity Constraints

First, we define the `NULL` value.

`NULL` represents an attribute whose value is unknown or does not exist for a particular entity instance. In general, it is a special marker indicating the absence of a value.

A **foreign key** is an attribute (or a set of attributes) that stores the value of the primary key of another relation.

Formally, a set of attributes FK in a relation schema R_1 is a foreign key if these attributes have the same domains as the attributes of the primary key in another relation schema R_2, and for each tuple $t_1 \in R_1$, the value of FK either matches the value of the primary key in some tuple $t_2 \in R_2$, or is `NULL`.

For each new tuple (record), the insertion is valid if:

1. The primary key value does not already exist in the relation.
2. Each foreign key value is either entirely `NULL` or entirely non-`NULL` and occurs in the referenced relation.

Now, assuming our dataset is originally in a valid state, how can it become invalid? Through **insertion**, **deletion**, or **update** operations.

There are three important **integrity constraints** in the relational model:

1. **Key constraint:** Candidate key values must be unique in every relation instance.

2. **Entity integrity:** An attribute that is part of a primary key cannot be `NULL`.

3. **Referential integrity:** Ensures that foreign key values in one relation correspond to existing primary key values in another relation.

If any of these integrity constraints are violated, the dataset enters an invalid state.

For example:

- Inserting a tuple with a duplicate primary key or without a primary key value violates the **key** or **entity integrity** constraints.
- Deleting a record that is referenced as a foreign key in another relation violates the **referential integrity** constraint.
- Updating a tuple's key value to one that causes duplication or breaks foreign key references can also result in an invalid state.

2.4 Relational Algebra

Query languages are specialized languages used to ask questions or queries about data in a database. **Relational algebra** is a **procedural** query language, defined as a step-by-step procedure for computing the desired result. Its operators take one or two relations as input and produce a new relation as output.

Queries in relational algebra are composed of a collection of such operators. In this chapter, we will learn six basic operators:

- **Select** (σ)
- **Project** (Π)
- **Union** ($\cup$)
- **Set difference** ($-$)
- **Cartesian product** ($\times$)
- **Rename** (ρ)

Before diving into relational algebra, we first review some fundamental set concepts.

2.4.1 Basic Set Concepts

Some basic set concepts:

- $S = \{a, b, c\}$ (A set S contains elements a, b, and c.)
- $a \in S$ (a is an element of the set S.)
- $a \notin S$ (a is not an element of the set S.)
- $\{\ x \in S \mid P(x)\ \}$ (The set of all x in S such that $P(x)$ is true.)

To describe relationships between sets A and B:

- A is a **subset** of B, denoted $A \subseteq B$, if and only if every element of A is also an element of B.
- A is **not a subset** of B, denoted $A \nsubseteq B$, if there exists at least one element of A that is not an element of B.
- A is **equal** to B, denoted $A = B$, if and only if $A \subseteq B$ and $B \subseteq A$.

2.4.2 Select Operation

For the **select** operation, we use the following notation:

$$\sigma_P(r) = \{\ t \mid t \in r \wedge P(t)\ \}$$

where P is called the **selection predicate**.

This operation can be understood as: from relation r, we select the tuples that satisfy the condition P.

The selection predicate P is a Boolean expression, which can involve multiple conditions using logical connectives, i.e., terms connected by $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT). Each term is of the form

`<attribute1> op <attribute2>` or `<attribute> op <constant>`,

where op $\in \{=, \neq, >, \geq, <, \leq\}$.

In practice, the selection operation σ specifies the tuples to retain based on the selection predicate.

2.4.3 Project Operation

For the **project** operation, we use the following notation:

$$\Pi_{A_1, A_2, \ldots, A_k}(r)$$

where $A_1, A_2, \ldots, A_k$ are attribute names.

This operation can be understood as: from relation r, we extract only the columns $A_1, A_2, \ldots, A_k$, while all other columns are projected out. Duplicate rows are eliminated in the final result.

Example. Consider the following relation **Employee**:

f_name	l_name	id	sex	salary	superid	dno
Joseph	Chan	999999	M	29500	654321	4
Victor	Wong	001100	M	30000	888555	5
Carrie	Kwan	898989	F	26000	654321	4
Joyce	Fong	345345	F	12000	777888	4

1. Find all employees who work in department 4 and whose salary is greater than 25000.

   Solution:

   $$\sigma_{\text{dno}=4 \wedge \text{salary}>25000}(\text{Employee})$$

2. Find the employee names and department numbers of all employees.

   Solution:

$$\Pi_{\text{f\_name, l\_name, dno}}(\text{Employee})$$

3. Find the sex and department number of all employees.

   Solution:

$$\Pi_{\text{sex, dno}}(\text{Employee})$$

4. Find the names of all employees who work in department 4 and whose salary is greater than 25000.

   Solution:

$$\Pi_{\text{f\_name, l\_name}}\left(\sigma_{\text{dno}=4 \wedge \text{salary}>25000}(\text{Employee})\right)$$

2.4.4 Union Operation

For the **union** operation, we use the following notation:

$$r \cup s = \{\ t \mid t \in r \vee t \in s\ \}$$

This operation returns a relation instance containing all tuples that occur in either relation R or relation S (or both). Duplicate tuples are eliminated in the result.

The schema of the result is identical to the schema of R, and the attribute names are inherited from R.

Intuitively, for $R \cup S$ to be valid, R and S must have the same arity (the same number of attributes), and their corresponding attribute domains must be **union-compatible**.

In general, two or more relations are union-compatible if they have the same number of attributes and the corresponding attributes have the same domains.

Remark. The union operation is **commutative**: $R \cup S = S \cup R$.

2.4.5 Set Difference Operation

For the **set difference** operation, we use the following notation:

$$r - s = \{\ t \mid t \in r \wedge t \notin s\ \}$$

This operation returns a relation instance containing all the tuples that occur in R but not in S.

Remark. Set differences must be taken between **union-compatible** relations.

Set difference is **not commutative**; in general, $R - S \neq S - R$.

2.4.6 Cartesian Product

For the **Cartesian-Product** operation, we use the following notation:

$$r \times s = \{\ pq \mid p \in r \wedge q \in s\ \}$$

This operation returns a relation instance whose schema contains all the fields of R followed by all the fields of S. The result contains one tuple $\langle r, s \rangle$ (concatenation of tuples r and s) for each pair of tuples $r \in R$, $s \in S$.

Example. We have two tables:

EMPLOYEE: f_name, l_name, id, bdate, addr, sex, salary, super_id, dno

DEPENDENT: eid, dep_name, sex, bdate, relationship (where eid is a foreign key referencing Employee)

Retrieve, for each female employee, a list of the names of her dependents.

Solution:

$$\text{Female\_emps} \leftarrow \sigma_{\text{sex} = \text{'F'}}(\text{Employee})$$

$$\text{Empnames} \leftarrow \Pi_{\text{f\_name, l\_name, id}}(\text{Female\_emps})$$

$$\text{Emp\_dependents} \leftarrow \text{Empnames} \times \text{Dependent}$$

$$\text{Actual\_dependents} \leftarrow \sigma_{\text{id=eid}}(\text{Emp\_dependents})$$

$$\text{Result} \leftarrow \Pi_{\text{f\_name, l\_name, dep\_name}}(\text{Actual\_dependents})$$

In the above example, the operator $\leftarrow$ is the **assignment operator**. The result of the expression on the right-hand side of $\leftarrow$ is assigned to the relation variable on the left-hand side.

This operator provides a convenient way to express complex queries, allowing us to write a query as a sequential program consisting of a series of assignments, followed by an expression whose value is displayed as the query result. Assignment must always be made to a temporary relation variable.

2.4.7 Rename Operation

For the **rename** operation, we use the following notation:

$$\rho_X(E)$$

This operation returns the expression E under the name X.

If a relational algebra expression E has arity n, then

$$\rho_{X(A_1, A_2, \ldots, A_n)}(E)$$

returns the result of expression E under the name X, with its attributes renamed to $A_1, A_2, \ldots, A_n$.

When dealing with Cartesian products, it is often necessary to use renaming.

Notation $w(W)$ implies that a relation w follows the schema W. It is assumed that the attributes of $r(R)$ and $s(S)$ are disjoint, i.e., $R \cap S = \varnothing$. If the attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming *must* be used.

When attribute names are the same in different relations, such as $r(A, B)$ and $s(A, C)$, we use **dot-notation** to distinguish them:

$$r.A \text{ and } s.A$$

This naming scheme avoids ambiguity during operations such as joins and products.

Remark. The **rename** (ρ) operator is a *basic* operator in relational algebra, while the **assignment** ($\leftarrow$) operator is not. Assignment simply allows us to store intermediate results under a temporary name for convenience — it does not modify the relations themselves or affect the algebraic computation.

Now we look at some additional operations.

2.4.8 Set-Intersection Operation

For the **set-intersection** operation, we use the following notation:

$$r \cap s = \{\ t \mid t \in r \wedge t \in s\ \}$$

This operation returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be **union-compatible**, and the schema of the result is defined to be identical to the schema of R.

The intersection operation can also be expressed using set difference:

$$r \cap s = r - (r - s)$$

2.4.9 Natural Join

For the **natural join** operation, we use the following notation:

$$r \bowtie s$$

The natural join $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows.

Consider each pair of tuples t_r from r and t_s from s. If t_r and t_s have the same values on each of the attributes in $R \cap S$, add a tuple t to the result, where t has the same values as t_r on attributes from r and t has the same values as t_s on attributes from s.

For example, let $R = (A, B, C, D)$ and $S = (E, B, D)$. Then, the result schema is (A, B, C, D, E). The natural join can be expressed as:

$$r \bowtie s = \Pi_{r.A, r.B, r.C, r.D, s.E}\big(\sigma_{r.B=s.B \wedge r.D=s.D}(r \times s)\big)$$

> **Remark.** If the two relations have no attributes in common, then the natural join is simply the Cartesian product. The natural join operation is both **associative** and **commutative**.

2.4.10 Division Operation

Consider two relations A and B, where A has exactly two fields x and y, and B has just one field y, with the same domain as the y field in A.
The division operation A/B returns the set of all x values (in the form of unary tuples) such that for every y value in a tuple of B, there exists a tuple $\langle x, y \rangle$ in A.
An analogy with integer division may help to understand this operation:

- For integers A and B, A/B is the largest integer Q such that $B \cdot Q \leq A$.
- For relation instances A and B, A/B is the largest relation instance Q such that $Q \times B \subseteq A$.

x	y
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
ε	1
ε	2

A

y
1
2

B

x
α
β

A/B

Computation using relational algebra:

$$A/B = \Pi_x(A) - \Pi_x\big((\Pi_x(A) \times B) - A\big)$$

Here $\Pi_x(A)$ extracts all x values from A. $(\Pi_x(A) \times B) - A$ finds all x values that do *not* pair with every y in B. Subtracting this from $\Pi_x(A)$ gives the final result of A/B.

> **Example.** We have two tables:
>
> **EMPLOYEE:** f_name, l_name, id, bdate, address, salary, sid, dno
>
> **WORKS_ON:** id, pno
>
> Retrieve the names of employees who work on all the projects that `John Sung` works on.
>
> **Solution:**
>
> $$\text{Sung} \leftarrow \sigma_{\text{f\_name = 'John'} \wedge \text{l\_name = 'Sung'}}(\text{EMPLOYEE})$$
> $$\text{Sung\_pnos} \leftarrow \Pi_{\text{pno}}(\text{WORKS\_ON} \bowtie \text{Sung})$$
> $$\text{Result\_id} \leftarrow \text{WORKS\_ON}/\text{Sung\_pnos}$$
> $$\text{Result} \leftarrow \Pi_{\text{f\_name, l\_name}}(\text{Result\_id} \bowtie \text{EMPLOYEE})$$

Example. Consider the relational schemas below:

Sailors(sid, sname, age)

Boats(bid, bname, color)

Reserves(sid, bid, date)

1. Find the names of sailors who have reserved the boat with bid = 103.
   Solution:
   $$\Pi_{\text{sname}}\big((\sigma_{\text{bid}=103}(\text{Reserves})) \bowtie \text{Sailors}\big)$$
2. Find the names of sailors who have reserved at least one red boat.
   Solution:
   $$\Pi_{\text{sname}}\big((\sigma_{\text{color}=\text{'red'}}(\text{Boats})) \bowtie \text{Reserves} \bowtie \text{Sailors}\big)$$
3. Find the names of sailors who have reserved at least a red or a green boat.
   Solution:
   $$\rho_{\text{Tempboats}}\big(\sigma_{\text{color}=\text{'red'}\vee\text{color}=\text{'green'}}(\text{Boats})\big)$$
   $$\Pi_{\text{sname}}\big(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors}\big)$$
4. Find the names of sailors who have reserved at least one red boat and at least one green boat.
   Solution:
   $$\rho_{\text{Tempred}}\big(\Pi_{\text{sid}}((\sigma_{\text{color}=\text{'red'}}(\text{Boats})) \bowtie \text{Reserves})\big)$$
   $$\rho_{\text{Tempgreen}}\big(\Pi_{\text{sid}}((\sigma_{\text{color}=\text{'green'}}(\text{Boats})) \bowtie \text{Reserves})\big)$$
   $$\Pi_{\text{sname}}\big((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors}\big)$$
5. Find the names of sailors who have reserved all boats.
   Solution:
   $$\rho_{\text{Tempsids}}\big((\Pi_{\text{sid,bid}}(\text{Reserves}))/(\Pi_{\text{bid}}(\text{Boats}))\big)$$
   $$\Pi_{\text{sname}}\big(\text{Tempsids} \bowtie \text{Sailors}\big)$$
6. Find the names of sailors who have reserved all red boats.
   Solution:
   $$\rho_{\text{Tempsids}}\big((\Pi_{\text{sid,bid}}(\text{Reserves}))/(\Pi_{\text{bid}}(\sigma_{\text{color}=\text{'red'}}(\text{Boats})))\big)$$
   $$\Pi_{\text{sname}}\big(\text{Tempsids} \bowtie \text{Sailors}\big)$$

2.4.11 Theta Join

For the **theta join** operation, we use the following notation:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

The theta join is defined by combining a selection and a Cartesian product into a single operation. It is the most general form of the join operation, where the *join condition* θ is any valid selection condition.

The comparison condition θ can involve any of the standard comparison operators:

$$=,\ \neq,\ >,\ <,\ \geq,\ \leq$$

When the join condition only involves equality ($=$), the operation is called an **equijoin**.

The **natural join** is a special case of the equijoin, in which equalities are specified on all attributes that have the same name in both relations R and S. In this case, the join condition can be omitted. The resulting schema contains all attributes of R, followed by the attributes of S that are not present in R.

Chapter 3

SQL