Chapter 1, Databases and Database Users

• Overview of the Course

- **What is a database ?**
  - General Definition: A collection of related data.
  - Restricted Definitions:
    - A database represents some aspects of real world.
    - It is a logical coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
    - It is designed, built and populated with data for an intended group of users and preconceived applications.
- A database may be generated and maintained manually or by machine. A computerized database may be created or maintained either by application programs written specifically for that task or by a Database Management System (DBMS).
- DBMS is a general-purpose software system that facilitates the processes of **defining**, **constructing**, and **manipulating** databases for various applications.
- **Definition:** specify data types, structures and constraints for the data.
- **Construction:** store the data itself on some storage medium that is controlled by DBMS.
- **Manipulation:** querying, updating, and generating reports.

• A University Database Example

- See Figure 1.2 (Figure 1.2 on e3)
- **Definition:** Specify the structure of the records of each file by specifying the different types of data elements (attributes) in each record. Specify a data type for each data element.
- **Construction:** Store data to represent each student, course, section, grade report, and prerequisite as a record in the appropriate file.

- **Manipulation:**
  - Queries: "retrieve a transcript of Smith", " list the names of students who took the section of database course offered in fall 1999 and their grades"
  - Updates: "change the class of Smith to sophomore", "create a new section for the database course for this semester"

• Characteristics of the Database Approach

- **Self-Describing Nature of the Database System.**

  complete description or defn of the database is stored in the system **catalog**, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. This information is referred to as **meta-data**.

  DBMS software should work equally well with any number of database applications - as long as the database definition is in the catalog.

- **Insulation between Programs and Data, and Data Abstraction**

  traditional file processing - structure of data files is embedded in access programs., so any change to the structure of a file may require changing all programs.

  DBMS access programs do not require such changes, since the structure of data files is stored in the DBMS catalog separately from the access programs (**program-data independence**).

  e.g. adding a new field to STUDENT file can be done easily in a DBMS by just changing the defn without changing the programs.

  Program-operation independence - Operation: interface and implementation. User application programs can operate on the data by invoking these operations through their interfaces, regardless of how they are implemented.

  Data abstraction - A DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored. Compare Fig 1.2 and 1.3 (Fig 1.2 and 1.3 on e3).

  Operation abstraction - e.g. Calculate-GPA.

– **Support of Multiple Views of Data**

A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored. e.g. Fig 1.4(a) (Fig 1.4 on e3) transcript view.

– **Sharing of Data and Multiuser Transaction Processing**

Sharing is essential if data for multiple applications is to be integrated and maintained in a single database. Thus, the DBMS must include **concurrency control**. e.g. reserving a seat on a airline - must be atomic.

• Advantages of Using a DBMS

– **Controlling redundancy**

File processing approach leads to redundancy.

* data has to be entered multiple times leading to duplication of effort.
* waste of storage space.
* data may easily become inconsistent.

With database approach, the views of different user groups are integrated during database design. This prevents inconsistency and it saves storage space.

Controlled Redundancy may be useful for improving the performance of queries. e.g. We may store StudentName and CourseNumber redundantly in a GRADE_REPORT file to avoid searching multiple files for frequently access data. The DBMS can automatically check:

* StudentName - StudentNumber in GRADE_REPORT matches one of Name - StudentNumber values of the STUDENT record.
* SectionId - CourseNo in GRADE_REPORT matches one in SECTION

See Fig. 1.5 (a) and (b) (Fig 1.5(a) and (b) on e3).

– **Restricting unauthorized access**

Security and authorization subsystem for DBMS.

Create accounts with varying privileges protected by passwords. Privileges in-

cludes which data item is allowed to access and what access operation is allowed to operate on that data item.

- **Persistent storage for program objects and data structure**

  To save complex program variables, programmer must explicitly store them in permanent files, which often involves converting these complex structures into format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file form at to the program variable structure.

  A complex object in C++ can be stored permanently in an object-oriented DBMS and can later be directly retrieved by another C++ program. DBMS automatically performs any necessary conversions.

- **Storage structures for efficient query processing**

  The database is stored on disk. The DBMS provides an efficient way to access data on disk using **indexes** (chapter 14).

  Given a query, the DBMS will choose an efficient execution plan (perform a sequence of operations in what order) based on the existing storage structures.

- **Providing backup and recovery**

  * restoring the state it was in before the failure causing program started executing (rollback)
  * or resume the execution from the point the program was interrupted

- **Providing multiple user interfaces**

  * query languages for casual users
  * programming language interfaces for application programmers
  * forms and command codes for parametric users. (GUI)
  * menu-driven interfaces for stand-alone users. (GUI)

- **Representing complex relationships among data**

  A DBMS must be capable of representing a variety of complex relationships among the data. Each SECTION record is related to one COURSE record as well as a

number of GRADE_REPORT records.

Exercises 1.9 - Name all relationships among records in Fig 1.2 (Fig 1.2 on e3).

- **Enforcing integrity constraints**
  - Specify a data type for each data item. e.g. value of Class is in [1,5]; Name $\leq$ 30 chars.
  - Every SECTION record must be related to a COURSE record
  - Every COURSE record must have a unique value for CourseNumber
  - Exercises 1.11: cite some examples of integrity constraints on the university database

• When Not to Use a DBMS ?

- DBMS has overhead.
  - high initial investments
  - too much generality
  - overhead for providing security, concurrency control, recovery and integrity
- May be more desirable to use regular files for
  - simple, well-defined, static applications
  - time critical applications. DBMS overhead may not make it real time.
  - multi-user access is not required

Chapter 2, Database System Concepts and Architecture

• Data Models, Schemas, and Instances

Data Model: A collection of concepts that can be used to describe the structure of a database. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database. Data models can also specify **behavior**; this allows the database designers to specify a set of **user-defined operations** in addition to basic operations.

- **Categories of Data Models**
  - **High-level or Conceptual Data Models** provide concepts that are close to the way many users perceive data. (e.g. ER Model use concepts such as entities, attributes, and relationships.)
    - An **entity** represents a real-world object or concept, such as an employee or a project.
    - 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.
  - **Representational or Implementation Data Models** provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer.
    - **Record-based:** relational, hierarchical and network
    - **Object-oriented:** in between conceptual and representational
  - **Physical Data Models** provide concepts that describe the details of how data is stored in the computer.
- **Schemas, Instances, and Database State**
  - A **Database Schema** is a description of the database. It is specified during database design and does not change frequently.

* A displayed schema is called a **Schema Diagram**, see Figure 2.1 (Figure 2.1 on e3). A schema diagram displays only some aspects of a schema.
* The data in the database at a particular time is called a **Database State** or the current set of **Instances**
* The DBMS is partly responsible for ensuring that every state of the database is a valid state.

• DBMS Architecture and Data Independence

– **The Three-Schema Architecture**: see Figure 2.2 (Figure 2.2 on e3).

* **Internal or Physical Schema:** The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
* **Conceptual Schema:** The conceptual schema uses high-level or implementation data model and hides the details of physical storage structure and concentrates on describing entities, data types, relationships, user operations and constraints.
* **External Schema or User View:** Each external schema uses high-level or implementation data model and describes the part of database that a particular user group is interested in and hides the rest of the database (security issues).

DBMS must transform a user request specified on an external schema into a request on the conceptual schema, then into a request on the internal schema for processing over the stored database. Also the results need to be transformed back to match the user's external view.

– **Data Independence:** The capacity to change the schema at one level of a database system without having to change the next higher level schema.

* **Logical Data Independence:** Changes of the conceptual schema should not affect the external schema.

  e.g. The external schema of Figure 1.4(a) (Figure 1.4(a) on e3) should not

be affected by changing the GRADE_REPORT file shown in Figure 1.2 (Fig 1.2 on e3) into the one shown in Figure 1.5(a) (Fig 1.5(a) on e3).

* **Physical Data Independence:** Changes of the physical schema should not affect the conceptual (or external) schemas.

Note that the changes of schema in one level do not affect the next higher level schema, but the **mapping/transformation** between these two level should be changed.

- # Database Languages

  - **Data Definition Language (DDL):** for defining Conceptual schema
  - **Storage Definition Language (SDL):** for defining internal schema
  - **View Definition Language (VDL):** for defining external views
  - **Data Manipulation Language (DML):** for defining retrieval, insertion, deletion, and modification of the data.

A comprehensive database language **SQL** represents a combination of DDL, VDL, and DML. Usually, the SDL is typically kept separate from the comprehensive language.

- # Database System Environment

see Figure 2.3 (Fig 2.3 on e3).

  - **DBMS Component Modules**
    * **Stored Data Manager:** control access to information that is stored on disk (database or catalog)
    * **DDL Compiler:** process schema definitions, specified in DDL, and store descriptions of the schemas in the DBMS catalog
    * **Run-time Database Processor:** handle database access at run time
    * **Query Compiler:** handle high-level queries and transform them to **access codes** and generate calls to the run-time processor for executing the codes.
    * **DML Compiler:** compile DML commands into object codes for database access

    - **Pre-compiler:** extract DML commands from a host programming language program and send them to the DML compiler.
  - **Database System Utilities**
    - **Loading:** A loading utility is used to load existing data file into the database. This utility can reformat the source file to desired database file and store it in the database.
    - **Backup:** dump the entire database into tape.
    - **File Reorganization:** reorganize a database file to gain performance
    - **Performance Monitoring:** monitor database usage and provide statistics to the DBA.

- # Classification of Database Management Systems

  several classification criteria, such as Data model, number of users, number of sites, cost. The main criterion (data model) is listed below.

  - **Relational Data model:** A database is a collection of tables, where each table can be stored as a separate file.
  - **Object Data model:** define a database in terms of objects, their properties, and their operations.
  - **Network Data Model:** represent data as **record types** and also represent a limited type of 1:N relationship, called a **set type**. See Figure 2.8 (Fig 2.4 on e3).
  - **Hierarchical Data Model:** represent data as hierarchical tree structures.

Chapter 3, Data Modeling Using the Entity-Relationship Model

• A Company Database Application Example

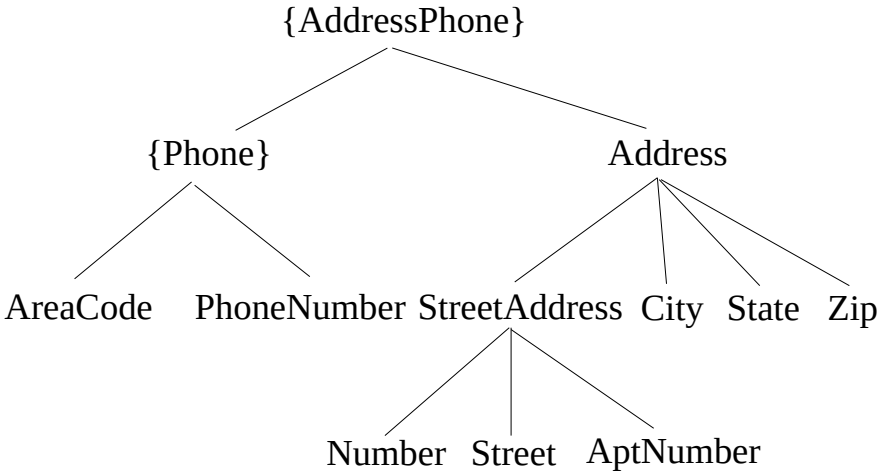
After the requirements collection and analysis phase, the database designers stated the following description of the Company

- The company is organized into **DEPARTMENTs**, Each department has a unique **Name**, a unique **Number**, and a particular **EMPLOYEE** who **MANAGES** the department. We keep track of the **Start Date** when that employee began managing the department. A department may have several **Locations**.
- A department **CONTROLs** a number of **PROJECTs**, each of which has a unique **Name**, a unique **Number**, and a single **Location**.
- We store each employee's **Name, Ssn, Address, Salary, Sex,** and **Birth Date**. An employee is **ASSIGNed** to one department but may **WORK ON** several projects, which are not necessarily **CONTROLLed** by the same department. We keep track of the number of **Hours** per week that an employee works on each project. We also keep track of the direct **SUPERVISOR** of each employee.
- We want to keep track of the **DEPENDENTs** of each employee for insurance purposes. We keep each dependent's **First Name, Sex, Birth Date,** and **Relationship** to the employee.
- Figure 3.2 (Fig 3.2 on e3) shows a possible **ER diagram** for this company database.

• Entity Types, Entity Sets, Attributes, and Keys

- **Entity:** An object in the real world. Physical existence - a particular person. Conceptual existence - a company, a university course.
- **Attributes:** The particular properties that describe an entity.
  - **Composite vs. Simple Attributes:** depend on whether attributes are divisible or not. e.g. Address(Street Address, City, State, Zip)

* **Single-Valued vs. Multivalued Attributes:** depend on how many values are allowed for an attribute. e.g. an Age attribute (single) for a person, an Address attribute (multiple) for a person.
* **Stored vs. Derived Attributes:** A derived attribute can be determined by a stored attribute. e.g. The Age attribute for a person can be derived from the Birth Date attribute of the person.
* **Null Value:** No applicable value or the value is unknown to an attribute. e.g. an ApartmentNumber attribute would have null value for a single-family home address. a CollegeDegrees attribute may have null value for a person.
* **Complex Attributes:** composite and multivalued attributes can be nested using () for composite and {} for multivalued attributes.
  e.g. {AddressPhone({Phone(AreaCode, PhoneNumber)},
  Address(StreetAddress(Number, Street, AptNumber), City, State, Zip))}



– **Entity Types:** define a collection of entities that have the same attributes. It is described by its name and attributes. See Figure 3.6 (Fig 3.6 on e3).
– **Entity Sets:** The collection of all entities of an entity type at any point in time. See Figure 3.6 (Fig 3.6 on e3).
– **Key Attributes:** The values of a key attribute are distinct for each individual entity in an entity set. Its value can be used to identify each entity uniquely.
  * **Simple Attribute Key**

* **Composite Attribute Key**

– **Value Sets (Domains) of Attributes:** specify the set of all possible values that may be assigned to an attribute for each individual entity. e.g. [1,5] for StudentClass in the university database example.

An attribute A of entity type E whose value set is V. Let $P(V)$ denotes the power set of V. Then $A : E \rightarrow P(V)$.

For a composite attribute A, the value set $V = P(V_1) \times P(V_2) \times \ldots \times P(V_n)$

– **Initial Conceptual Design of the COMPANY Database without Relationship Types**

DEPARTMENT Name, Number, {Locations}, Manager, ManagerStartDate
PROJECT Name, Number, Location, ControlDepartment
EMPLOYEE Name(FName, MInit, LName), SSN, Sex, Address, Salary BirthDate, Department, Supervisor, {WorkOn(Project, Hours)}
DEPENDENT Employee, DependentName, Sex, BirthDate, Relationship

• Relationships, Relationship Types, Roles, and Structural Constraints

– **Relationship Types:** A mathematical relation on n entity types $E_1, E_2, \ldots, E_n$, or it can be defined as a subset of the Cartesian product $E_1 \times E_2 \times \ldots \times E_n$.

– **Relationship Sets:** A set of **relationship instances** r_i, where each r_i associates n individual entities $(e_1, e_2, \ldots, e_n)$. See Figure 3.9 (Fig 3.9 on e3).

– **Relationship Degree:** number of participating entity types on the relation type. Figure 3.9 (Fig 3.9 on e3) shows binary degree; Figure 3.10 (Fig 3.10 on e3) shows ternary degree.

– **Relationships $\longleftrightarrow$ Attributes:**

    * **Relationships as Attributes:** WORKS_FOR relationship type between EMPLOYEE and DEPARTMENT $\longrightarrow$ a Department attribute of the EMPLOYEE or a multivalued Employees attribute of the DEPARTMENT.
    * **Attributes as Relationships:** If an attribute of an entity type refers to another entity type, then this attribute can be changed to a relationship type between these two entity type.

- **Role Names:** Each entity type that participates in a relationship type plays a particular **role** in the relationship. Role name is necessary in a situation that the same entity type participates more than once in a relationship type (**recursive relationship**) in different roles. e.g. SUPERVISION relationship type in Figure 3.11 (Fig 3.11 on e3).
- **Relationship Constraints:**
    * **Cardinality Ratios for Binary Relationships:** specify the number of relationship instances that an entity can participate in. Possible Ratios are 1:1, 1:N, M:N. e.g. MANAGES, WORKS_FOR, WORKS_ON relationship types in Figure 3.12, 3.9, 3.13 (Fig 3.12, 3.9, 3.13 on e3).
    * **Participation Constraints or Existence Dependencies:** specify whether the existence of an entity depends on its being related to another entity via the relationship type.
        · **Total Participation Constraints:** An entity can exist only if it participates in a relationship instance. e.g. If a company policy states that every employee must works for a department, then the participation of EMPLOYEE in WORKS_FOR is total participation.
        · **Partial Participation Constraints:** Some entities (not necessary all) participate in a relationship type. e.g. EMPLOYEE entity type to the MANAGES relationship type is a partial participation.
- **Attributes of Relationship Types:** Relationship type can also have attributes, similar to those of entity types.
    * Attributes of 1:1 relationship types can be migrated to either one of the par-

ticipating entity types.

e.g. StartDate of MANAGES can be migrated to StartDateOfManaging attribute of EMPLOYEE or ManagerStateDate attribute of DEPARTMENT.

* Attributes of 1:N relationship types can be migrated only to the entity type at the N-side of the relationship.

  e.g. StartDate of WORKS_FOR can be migrated to StartDate attribute of EMPLOYEE.

- **Weak Entity Type:** Entity types that do not have key attributes of their own. e.g. DEPENDENT in Figure 3.2 (Fig 3.2 on e3).
  - Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with some of their attributes values.
  - **Identifying or Owner Entity Type:** The owner entity type can be used to identify an entity of a weak entity type.
  - **Identifying Relationship Type:** relate a weak entity type to its owner entity type.
  - A weak entity type always has a total participation to its identifying relationship type.
  - **Partial Key:** A set of attributes that can uniquely identify weak entities that are related to the same owner entity.
- **Notations for ER Diagrams:** See Figure 3.14 (Fig 3.14 on e3).
- **Refining the ER Design for the COMPANY Database with Relationship Types:** See Figure 3.2 (Fig 3.2 on e3).

DEPARTMENT Name, Number, {Locations}, Manager, ManagerStartDate
PROJECT Name, Number, Location, ControlDepartment
EMPLOYEE Name(FName, MInit, LName), SSN, Sex, Address, Salary BirthDate, Department, Supervisor, {WorkOn(Project, Hours)}
DEPENDENT Employee, DependentName, Sex, BirthDate, Relationship

Table 1: The original design without relationship types

New R Types	Participating E Types	Ratio	Participation	Attributes
MANAGES	DEPARTMENT, EMPLOYEE	1:1	Total, Partial	StartDate
WORKS_FOR	DEPARTMENT, EMPLOYEE	1:N	Total, Total	
CONTROLS	DEPARTMENT, PROJECT	1:N	Partial, Total	
SUPERVISION	EMPLOYEE, EMPLOYEE	1:N	Partial, Partial	
WORKS_ON	EMPLOYEE, PROJECT	M:N	Total, Total	Hours
DEPENDENTS_OF	EMPLOYEE, DEPENDENT	1:N	Partial, Total	

Table 2: The refined design with relationship types

- **Design Choices for ER Conceptual Design:**
  - An attribute is a reference to another entity type $\longrightarrow$ Relationship type.
    e.g. the Manager in DEPARTMENT in Figure 3.8 (Fig 3.8 on e3).
  - An attribute that exists in several entity types $\longrightarrow$ Its own entity type.
  - An entity type with a single attribute that is related to only one other entity type $\longrightarrow$ An attribute of the other entity type.
    e.g. If a DEPARTMENT with a single attribute DeptName and related to only one other STUDENT entity type $\longrightarrow$ An attribute Department of STUDENT.
  - More in Chapter 4 concerning **specialization/generalization** and relationships of higher degree.

Exercise 3.21

Design an ER schema for keeping track of information about votes taken in the U.S. House of Representives during the current two-year congressional session. The database needs to keep track of each U.S. State's Name (e.g., Texas, New York, California) and includes the Region of the State (domain is {Northeast, Midwest, Southeast, Southwest, West}). Each congressperson in the House of Representives is described by their Name, and includes the District represented, the StartDate when they were first elected, and the political Party they belong to (domain is {Republican, Democrat, Independent, Other}). The database keeps track of each Bill, and includes the BillName, the DataOfVote on the bill, whether the bill PassOrFailed (domain is {Yes, No}), and the Sponsor (the congressperson(s) who sponsored the bill). The database keeps track of how each congressionperson voted on each bill (domain is {Yes, No, Abstain, Absent}). Draw an ER schema diagram for the above application. State clearly any assumption you make.

Chapter 4, Enhanced Entity-Relationship Modeling

4.1 Subclasses, Superclasses, and Inheritance

- A **Subclass** S of an entity type C is a subset of C, i.e., $S \subseteq C$.
- We call C a **superclass** of the subclass S.
- All entities in S have some similar characteristics that all other entities in C S do not possess.
- An entity in a subclass and the corresponding entity in a superclass refer to the same real-world object.
- An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass.
- For example, an EMPLOYEE entity type can be grouped further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, ...

  Each of these subgroupings above is a subclass of EMPLOYEE, and EMPLOYEE is a superclass of each of these subclasses.
- We call the relationship between a superclass and any one of its subclasses a **superclass/subclass (C/S in short) or IS-A relationship**.
  e.g. EMPLOYEE/SECRETARY, EMPLOYEE/ENGINEER.
- An entity that is member of a subclass **inherits** all the attributes of the entity as a member of the superclass and all the relationships in which the superclass participates.
  See Figure 4.1 (Figure 4.1 on e3).

4.2 Specialization and Generalization

- **Specialization** is the process of defining a set of subclasses of an entity type.

- There should have some distinguishing characteristic of the entities in the superclass for the specialization process that specializes entities in the superclass into a set of subclasses.
  e.g. EMPLOYEE $\longrightarrow$ {SECRETARY, ENGINEER, TECHNICIAN} is based on the **job type** of each entity.

- The C/S relationship should be 1:1 relationship. The main difference is that in a 1:1 relationship type two distinct entities are related, whereas in the C/S relationship only one real-world entity is involved but playing in different roles.

- Two reasons for including C/S relationships and specializations/generalizations in the data model.
  - Certain attributes may apply to some but not all entities of the superclass.
  - Some relationship types may be participated in only by entities that are members of the subclass.
  - See example in Figure 4.1 (Fig 4.1 on e3).

- **Generalization** is the reverse process of specialization. For several entity types, we suppress the differences among them, identify their common features, and generalize them into a single superclass. e.g. See Figure 4.3 (Fig 4.3 on e3).

4.3 Constraints and Characteristics of Specialization and Generalization

- A superclass may have several specializations. e.g. EMPLOYEE $\longrightarrow$ {SECRETARY, TECHNICIAN, ENGINEER} and {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE} and {MANAGER}.

- If there is a condition on values for some attributes of the superclass that can determine exactly which entities are members of a subclass. This kind of subclasses are called **predicate-defined** subclasses. Also, the condition is the **defining predicate** of the subclass.

For example, If the EMPLOYEE entity type has a JobType attribute, we can specify the condition of membership for the SECRETARY subclass by the predicate (JobType = "Secretary") in the EMPLOYEE entity type. See Figure 4.4 (Fig 4.4 on e3).

- If all subclasses in a specialization have their membership condition on the same attribute of the superclass, then we call this specialization an **attribute-defined specialization**. Also, we call the attribute a **defining attribute** of the specialization.

- **Disjointness Constraint:** specify that the subclasses of the specialization must be disjointed.

- **Completeness Constraint:**

  - **Total:** specify that every entity in the superclass must be a member of some subclasses.
    e.g. EMPLOYEE $\longrightarrow$ {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.

  - **Partial:** allow an entity not to belong to any of the subclasses. e.g. EMPLOYEE $\longrightarrow$ {MANAGER}.

  - A superclass that was identified through the generalization process usually is total.

- **Specialization/Generalization Hierarchy:** Every subclass participates as a subclass in only one C/S relationship. See Figure 4.1 (Fig 4.1 on e3).

- **Specialization/Generalization Lattice:** A subclass can be a subclass in more than one C/S relationship. See Figure 4.6, 4.7 (Fig 4.6, 4.7 on e3).

- A subclass inherits the attributes not only of its direct superclass but also of all its predecessor superclasses all the way to the root of the hierarchy or lattice.

- A subclass with more than one superclass is called a **shared subclass**.

4.4 Modeling of UNION Types Using Categories

- **Category:** A subclass is a collection of objects that is a subset of the UNION of distinct entity types. This is used for modeling a single C/S relationship with more

than one superclass.

e.g. {PERSON, BANK, COMPANY} $\longrightarrow$ VEHICLE_OWNER

- Compare ENGINEERING_MANAGER in Figure 4.6 (Fig 4.6 on e3) to OWNER in Figure 4.8 (Fig 4.8 on e3).

  ENGINEERING_MANAGER is a subset of the intersection of three superclasses; whereas Owner is a subset of the union of three superclasses.

- The inheritance of a category vs. the inheritance of a shared subclass.

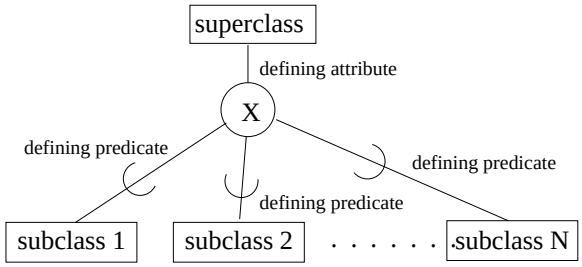
  OWNER in Figure 4.8 (Fig 4.8 on e3) vs. ENGINEERING_MANAGER in Figure 4.6 (Fig 4.6 on e3).

- Compare REGISTERED_VEHICLE category in Figure 4.8 (Fig 4.8 on e3) to the generalized superclass VEHICLE in Figure 4.3(b) (Fig 4.3(b) on e3).

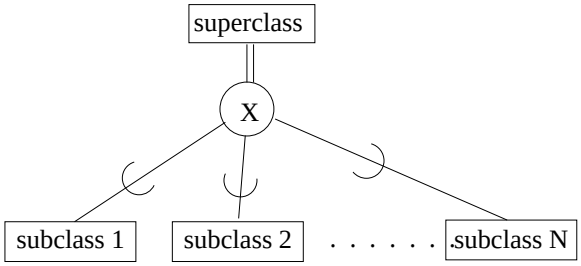
- A category can be **total** or **partial**. See Figure 4.9 on e3.

- If a category is total, it may be represented alternatively as a specialization. See Figure 4.9(b) on e3.

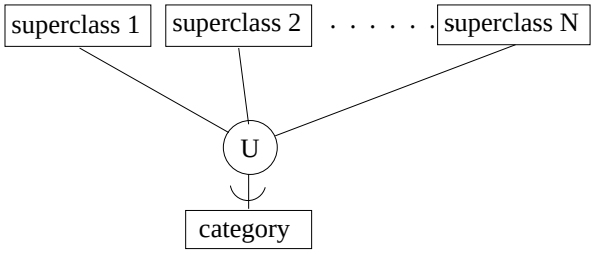
- **The Notation for EER Model:** See next page.



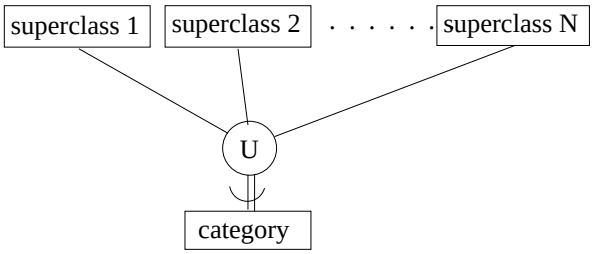
1. Partial Specialization/Generalization
2. X = ’d’ means Disjoint
3. X = ’o’ means overlap
4. The subset sign indicates the direction of a C/S relationship



Total Specialization/Generalization



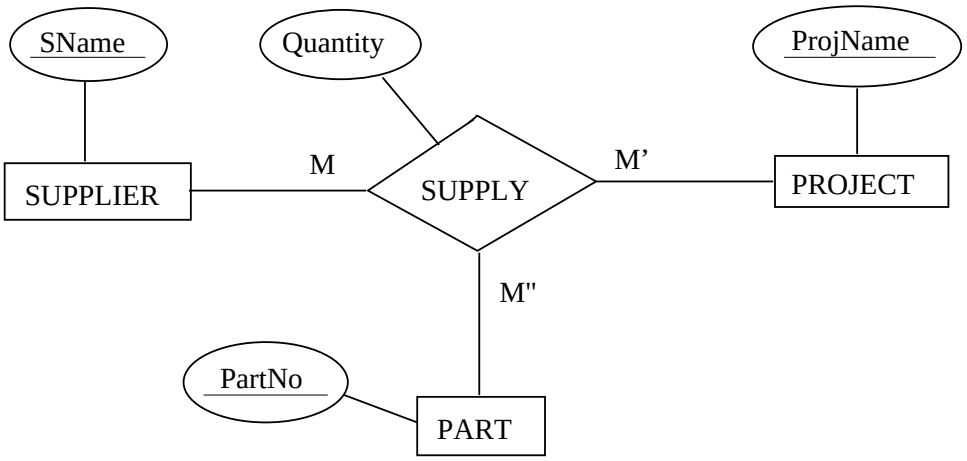
1. Partial Category
2. U means UNION



Total Category

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

- A ternary relationship type SUPPLY represents the relationship among SUPPLIER, PROJECT, and PART entity types.



Some suppliers supply some parts to some projects

  - Suppose we have SUPPLY relationship type instances as follows.

$$(s_1, j_1, p_1), (s_1, j_2, p_1), (s_1, j_1, p_2)$$

$$(s_2, j_1, p_1), (s_2, j_3, p_1), (s_2, j_2, p_3), (s_2, j_3, p_3)$$

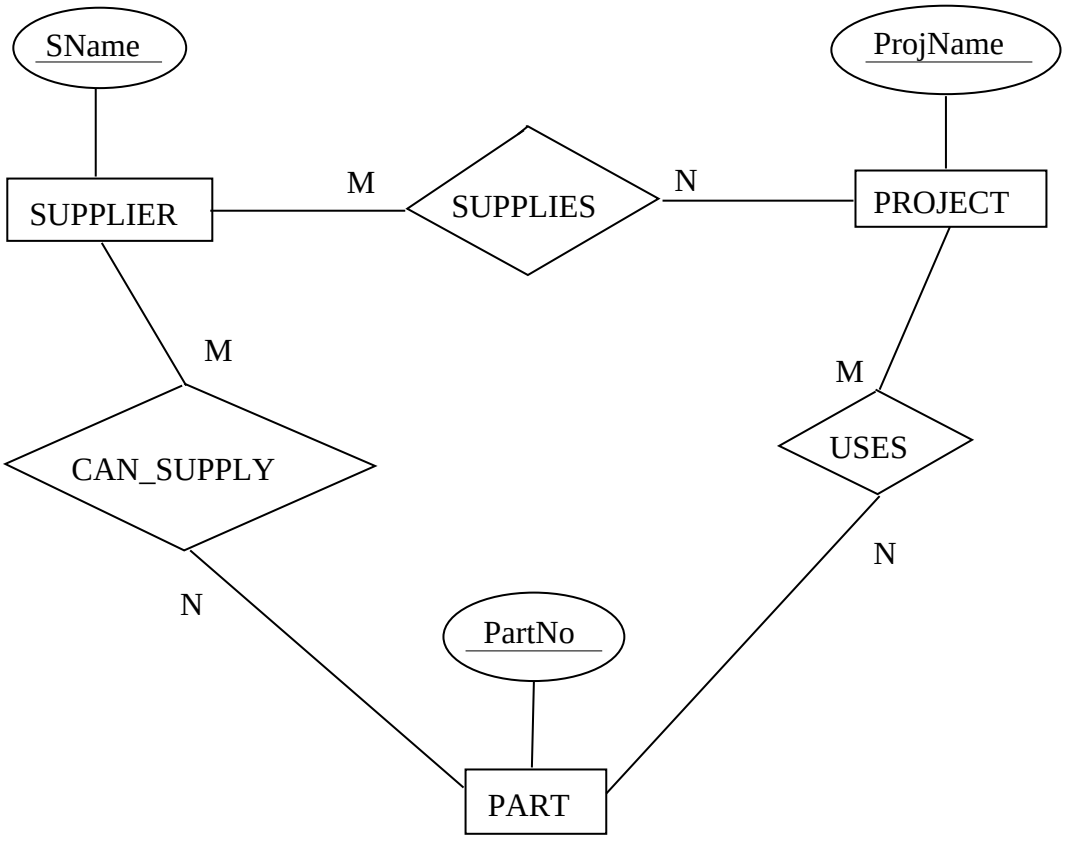
    For each (s_x, j_y) combination, we have multiple p_z
    For each (j_y, p_z) combination, we have multiple s_x
    For each (s_x, p_z) combination, we have multiple j_y

    $\Longrightarrow$ means the cardfinality ratio for the above ER-diagram should be $M : M' : M''$

  - If the schema states that for a particular project-part combination, only one supplier will be used. Then we put 1 on the SUPPLIER participation, which means that any combination of (j, p) can appear at most once in the relationship set. Thus, (j, p) could be a key for the relationship set.

- A ternary relationship type represents more information than three binary relationship types.

  Consider the following ER-diagram with three binary relationship types and compare it to the one (above) with one ternary relationship.



  - (s,p) & (j,p) & $(s,p) \not\Longrightarrow (s,j,p)$, but
    $(s,j,p) \Longrightarrow (s,p)$ & (j,p) & (s,p)
  - for example,

SUPPLIES	s_1	s_2
j_1	√	√
j_2		√

CAN_SUPPLY	s_1	s_2
p_1	√	√
p_2	√	√

USES	j_1	j_2
p_1	√	
p_2		√

    (s_1,j_1) & (s_1,p_1) & (j_1,p_1), but (s_2,p_1,j_1)

- If ternary relationship is not allowed, then change ternary relationship type to a weak entity type with three identifying relationship types as shown in Figure 4.11(c) (Fig 4.13(c) on e3).

Chapter 5, The Relational Data Model and Relational Database Constraints

5.1 Relational Model Concepts

- Represent the database as a collection of relations.
- Relation is a table of values.
- Each row represents an entity or relationship.
- table $leftrightarrow$ relation; row $leftrightarrow$ tuple; column header $leftrightarrow$ attribute.

5.1.1 Domains, Attributes, Tuples, and Relations

- A relation schema $R(A_1, A_2, \ldots, A_n)$ with corresponding set of domains $(D_1, D_2, \ldots, D_n)$, where $dom(A_i) = D_i$ for $1 \leq i \leq n$. The **degree** of R is n (the # of attributes).
- A relation (or **relation state**) $r(R) = \{t_1, t_2, \ldots, t_m\}$ is a set of n-tuples, where each $t =< v_1, v_2, \ldots, v_n >$ and each $v_i \in dom(A_i) \cup NULL$.
  $t[A_i]$: the i^{th} value in the tuple t.
- $r(R) \subseteq \underline{(dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n))}$
  The under-lining expression above is all possible combination of values for a relation with degree n.

5.1.2 Characteristics of Relations

- There is no ordering of tuples in a relation.
- Ordering of values (attributes) within a tuple depends on the formal definition.
- First Normal Form: each value in a tuple is **atmoic** (no composite and multivalued attributes).
  Composite $\longrightarrow$ simple component attributes.
  Multivalued $\longrightarrow$ separate relations.

- Entity type and relationship type are represented as relations in relational model.
  e.g. STUDENT relation of Figure 5.1 (Fig 7.1 on e3) and a relation schema MAJORS (studentSSN, DepartmentCode)

5.1.3 Relational Model Notation

- $R(A_1, A_2, \ldots, A_n)$: a relation schema R of degree n.
- $t =< v_1, v_2, \ldots, v_n >$: an n-tuple t in $r(R)$.
- $t[A_i]$ or $t.A_i$: value v_i in t for attribute A_i.
- $t[A_u, A_w, \ldots, A_z]$ or $t.(A_u, A_w, \ldots, A_z)$: subtuple of values $< v_u, v_w, \ldots, v_z >$ from t for attributes $A_u, A_w, \ldots, A_z$.
- Q, R, S : relation names.
- q, r, s : relation states.
- t, u, v : tuples.
- $R.A$: attribute A in a relation schema R.

5.2 Relational Model Constraints and Relational Database Schemas

5.2.1 Domain Constraints

- **Domain Constraint** : all $t[A_i] \in dom(A_i)$.

5.2.2 Key Constraints and Constraints on NULL

- A **superkey** SK of a schema R that is a subset of attributes that $t_1[SK] \neq t_2[SK]$.
- Default superkey : the set of all attributes.
- A key is a **minimal** superkey, i.e., you can not remove any attribute from the key and still make it as a superkey.

- If there are several keys in a relation schema, each of the keys is called a **candidate** key. See Figure 5.4 (Fig 7.4 on e3).

- Pick one from all candidate keys as the primary key.
  Criterion for choosing a primary key: smallest # of attributes.

5.2.3 Relational Databases and Relational Database Schemas

- A **relational database schema** $S = \{R_1, R_2, \ldots, R_m\}$ and a set of **integrity constraints**.

- A **relational database state** $DB = \{r_1, r_2, \ldots, r_m\}$ for each $r_i(R_i)$ such that each r_i satisfies integrity constraints.

- See Figure 5.5 and 5.6 (Fig 7.5 and 7.6 on e3).

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

- **Entity integrity constraint**: No primary key value can be NULL.

- **Referential integrity constraint**: A tuple in one relation that refers to another relation must refer to an existing tuple.

- A set of attributes $FK = \{A_u, A_v, \ldots, A_z\}$ in R is a foreign key of R_1 that references R_2 if

  (i) $dom(A_u) = dom(A_{u'})$, $dom(A_v) = dom(A_{v'}), \ldots, dom(A_z) = dom(a_{z'})$, where $PK = \{A_{u'}, A_{v'}, \ldots, A_{z'}\}$ in R_2.

  (ii) For any t_1 in R_1, $\exists$ some t_2 in R_2 such that $t_1[FK] = t_2[PK]$ or $t_1[FK] =< NULL, NULL, \ldots, NULL >$

  These two conditions of foreign key specify the referential integrity constraint formally. We call R_1 referencing relation, R_2 referenced relation.

- A foreign key can refer to its own relation. See Figure 5.5 (Fig 7.5 on e3), the SUPER-SSN in EMPLOYEE.

- The diagrammatic representation of the referential integrity constraint: see Figure 5.7 (Fig 7.7 on e3).

5.3 Update Operations and Dealing with Constraint Violations

5.3.1 The Insert Operation

- INSERT a tuple $t =< v_1, v_2, \ldots, v_n >$ to a relation R with degree n.
  - It may violate domain, key, entity integrity and referential integrity constraints (see examples in pp. 209).
  - If there is a violation, either reject the insertion or try to correct the reason for rejection.

5.3.2 The Delete Operation

- DELETE a tuple (tuples) t from R if t satisfies the (selection) condition.
  - (selection) condition : $< Clause >< Boolean\ OP >< Clause >< Boolean\ OP >$ ...
  - Clause : $< Attribute\ Name >< Comparison\ OP >< Const\ Value >$ or $< Attribute\ Name >< Comparison\ OP >< Attribute\ Name >$
  - Deletion may violate referential integrity constraint.
  - If there is a violation, three options: rejection, cascade the deletion, or modify the referential attribute value.

5.3.3 The Update Operation

- UPDATE the values of some attributes in a tuple (tuples) t in R if t satisfies the (selection) condition.
- Updating an attribute that is
  - a primary key: similar to delete a tuple, then insert another tuple.
  - a foreign key: may violate referential integrity or domain constraints.
  - otherwise, may violate domain constraint.

Chapter 6, The Relational Algebra and Relational Calculus

6.1 Unary Relational Operations: SELECT and PROJECT

6.1.1 The SELECT Operation

- SELECT a subset of tuples from R that satisfy a selection condition.
  - $\sigma_{<selection\ condition>}(R_1)$
  - $\sigma_{(DNO=4\ and\ SALARY>25000)\ or\ (DNO=5\ and\ SALARY>30000)}(EMPLOYEE)$
    See Figure 6.1(a) (Fig 7.8(a) on e3) for the result.
  - The resulting relation R_2 after applying selection on R_1, we have

    $$degree(R_1) = degree(R_2)\ and$$

    $$|\ R_2\ | \leq |\ R_1\ |\ \ for\ any\ selection\ condition$$
  - Commutative: $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$
  - $\sigma_{C_1}(\sigma_{C_2}(\ldots(\sigma_{C_n}(R))\ldots)) = \sigma_{C_1\ and\ C_2\ and.....and\ C_n}(R)$

6.1.2 The PROJECT Operation

- PROJECT some columns (attributes) from the table (relation) and discard the other columns.
  - $\pi_{<attribute\ list>}(R_1)$
  - $\pi_{LNAME,\ SALARY}(EMPLOYEE)$
  - R_2 has only the attributes in $<$ *attribute list* $>$ with the same order as they appear in the list.
  - $degree(R_2) = |<$ *attribute list* $>|$
  - PROJECT operation will remove any duplicate tuples (**duplication elimination**). This happens when attribute list contains only non-key attributes of R_1.

- $\mid R_2 \mid \leq \mid R_1 \mid$. If the $<$ *attribute list* $>$ is a superkey of R_1, then $\mid R_2 \mid = \mid R_1 \mid$
- $\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$

6.1.3 Sequences of Operations and the RENAME Operation

- It may apply several relational algebra operations one after another to get the final result.
  - Either we can write the operations as a single **relational algebra expression** by nesting the operations, or
  - we can apply one operation at a time and create intermediate result relations.
- For example, the algebra expression $\pi_{FNAME,\ LNAME,\ SALARY}(\sigma_{DNO=5}(EMPLOYEE))$ is equivalent to

$$\begin{cases} DEP5\_EMPS \longleftarrow \sigma_{DNO=5}\ (EMPLOYEE) \\ RESULT \longleftarrow \pi_{FNAME,\ LNAME,\ SALARY}(DEP5\_EMPS) \end{cases}$$

- We could rename the above intermediate (or final) relations by

$$\begin{cases} TEMP \longleftarrow \sigma_{DNO=5}(EMPLOYEE) \\ R(FN,\ LN,\ SALARY) \longleftarrow \pi_{FNAME,\ LNAME,\ SALARY}(TEMP) \end{cases}$$

  or we can define a RENAME operation and rewrite the query as follows.
  - $\rho_{S(B_1,B_2,...,B_n)}(R)$ or
  - $\rho_S(R)$ or
  - $\rho_{(B_1,B_2,...,B_n)}(R)$
  - Where S is the renamed relation name of R and B_i's are the renamed attribute names of R.
  - The query becomes

$$\begin{cases} \rho_{TEMP}(\sigma_{DNO=5}(EMPLOYEE)) \\ \rho_{R(FN,\ LN,\ SALARY)}(\pi_{FNAME,\ LNAME,\ SALARY}(TEMP)) \end{cases}$$

6.2 Relational Algebra Operations from Set Theory

6.2.1 The UNION, INTERSECTION, and MINUS Operations

- $R_1 \cup R_2$ (UNION), $R_1 \cap R_2$ (INTERSECTION), or $R_1 - R_2$ (SET DIFFERENCE) are valid operations iff R_1 and R_2 are **union compatible**.
- Two relations $R_1(A_1, A_2, \ldots, A_n)$ and $R_2(B_1, B_2, \ldots, B_n)$ are union compatible if $degree(R_1) = degree(R_2)$ and $dom(A_i) = dom(B_i)$ for $1 \leq i \leq n$.
- The resulting relation has the same attribute names as the first relation R_1
- Commutative: UNION, INTERSECTION
- Associative: UNION, INTERSECTION
- See Figure 6.4 (or Fig 7.11 on 3e)

6.2.2 The CARTESIAN PRODUCT (or CROSS PRODUCT) Operation

- $R_1 \times R_2$ (R_1 and R_2 do not need to be union compatible)
- $R_1(A_1, A_2, \ldots, A_n) \times R_2(B_1, B_2, \ldots, B_m) = Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, where
  - $| Q | = | R_1 | \times | R_2 |$
  - For example, or see Figure 6.5 (Fig 7.12 on e3)

B_1

A_1	A_2
a_{11}	a_{21}
a_{12}	a_{22}

$\times$ B_2

B_1
b_{11}
b_{12}
b_{13}

$=$ Q

A_1	A_2	B_1
a_{11}	a_{21}	b_{11}
a_{11}	a_{21}	b_{12}
a_{11}	a_{21}	b_{13}
a_{12}	a_{22}	b_{11}
a_{12}	a_{22}	b_{12}
a_{12}	a_{22}	b_{13}

6.3 Ninary Relational Operations: JOIN and DIVISION

6.3.1 The JOIN Operation

- JOIN is used to combine related tuples from two relations into single tuples.
- $R_1(A_1, A_2, \ldots, A_n) \bowtie_{<join\ condition>} R_2(B_1, B_2, \ldots, B_m) = Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$; where each tuple in Q is the combination of tuples – one from R_1 and one from R_2 – whenever the combination satisfies the join condition.
- $R_1 \bowtie_{<condition>} R_2 \equiv \sigma_{<condition>}(R_1 \times R_2)$
  - For example, see Figure 6.6 (Figure 7.13 on e3),

    ACTUAL_DEPENDENT $\longleftarrow$ EMPLOYEE $\bowtie_{SSN=ESSN}$ DEPENDENT

    $\equiv$ACTUAL_DEPENDENT $\longleftarrow$ $\sigma_{SSN=ESSN}$ (EMPLOYEE $\times$ DEPENDENT)
- Usually the join condition is of the form:

  $$< A_{i_1}\theta B_{j_1} > \ AND \ < A_{i_2}\theta B_{j_2} > \ AND \ldots AND \ < A_{i_p}\theta B_{j_p} >$$

  Each attribute pair in the condition must have the same domain; θ is one of the comparison operator.

6.3.2 The EQUIJOIN and NATURAL JOIN Variations

- All JOIN operations with only "=" operator used in the conditions are called EQUI-JOIN.
- Each tuple in the resulting relation of an EQUIJOIN has the same values for each pair of attributes listed in the join condition.
- **NATUAL JOIN** ($*$) was created to get rid of the superfluous attributes in an EQUI-JOIN.
  - NATUAL JOIN requires each pair of join attributes have the same name in both relations, otherwise, a renaming operation should be applied first.
  - For example, see Figure 5.6 and 6.7 (Figure 7.6 and 7.14 on e3),

    DEPT_LOCS $\longleftarrow$ DEPARTMENT $*$ DEPT_LOCATIONS

    PROJ_DEPT $\longleftarrow$ $\rho_{(DNAME,\ DNUM,\ MGRSSN,\ MGRSTARTDATE)}$ (DEPARTMENT)

$*$ PROJECT

- $0 \leq \mid R_1(A_1, A_2, \ldots, A_n) \bowtie_{<COND>} R_2(B_1, B_2, \ldots, B_m) \mid \leq n \times m$

6.3.3 A Complete Set of Relational Algebra Operations

- $\{\sigma, \pi, \cup, -, \times\}$ is a complete set; that is, any of the other relational algebra operations can be expressed as a sequence of operations from this set. For example,
  - $R \cap S \equiv (R \cup S) - ((R - S) \cup (S_R))$
  - $R \bowtie_{<COND>} S \equiv \sigma_{<COND>}(R \times S)$
  - A NATUAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.

6.3.4 The DIVISION Operation

- $T(Y) \longleftarrow R(Z) \div S(X)$, where X, Y, Z are sets of attributes and $X \subseteq Z$ and $Y = Z - X$
- A tuple $t \in T$ if tuples $t_1 \in R$ with $t_1[Y] = t$ and with $t_1[X] = t_2$ for every tuple t_2 in S. See Figure 6.8(b) (Figure 7.15(b) on e3).
- Example for DIVISION operation: "Retrieve the names of employees who work on all the projects that 'John Smith' works on.
  JSMITH_SSN(ESSN) $\longleftarrow \pi_{SSN}$ ($\sigma_{FNAME='John' \ AND \ LNAME='Smith'}$ (EMPLOYEE))
  JSMITH_PROJ $\longleftarrow \pi_{PNO}$ (JSMITH_SSN $*$ WORKS_ON)
  WORKS_ON2 $\longleftarrow \pi_{ESSN,PNO}$ (WORKS_ON)
  DIV_HERE(SSN) $\longleftarrow$ WORKS_ON2 $\div$ JSMITH_PROJ
  RESULT $\longleftarrow \pi_{FNAME, \ LNAME}$ (EMPLOYEE $*$ DIV_HERE)
  See figure 6.8(a) (Figure7.15(a) on e3).

6.4 Additional Relational Operations

6.4.1 Aggregate Functions and Grouping

- Apply aggregate functions **SUM, AVERAGE, MAXIMUM, MINIMUM** and **COUNT** of an attribute to different groups of tuples.

- $R_2 \longleftarrow {}_{<grouping\ attribs>}\Im_{<<func\ attrib>,<func\ attrib>,.....>}(R_1)$

  - The resulting relation R_2 has the **grouping attributes + one attribute for each element in the function list**.

  - For example,

    * ${}_{DNO}\Im_{COUNT\ SSN,\ AVERAGE\ SALARY}(EMPLOYEE)$

DNO	COUNT_SSN	AVERAGE_SALARY
5	4	33250
4	3	31000
1	1	55000

    * $\Im_{COUNT\ SSN,\ AVERAGE\ SALARY}(EMPLOYEE)$

COUNT_SSN	AVERAGE_SALARY
8	35125

6.4.3 OUTER JOIN

- **LEFT OUTER JOIN:** $R_3(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m) \longleftarrow R_1(A_1, A_2, \ldots, A_n) \ \supset\!\bowtie_{<JOINCOND>} R_2(B_1, B_2, \ldots, B_m)$

  - This operation keeps every tuple t in left relation R_1 in R_3, and fills "NULL" for attributes $B_1, B_2, \ldots, B_m$ if the join condition is not satisfied for t.

  - For example,
    $TEMP \longleftarrow (EMPLOYEE \ \supset\!\bowtie_{SSN=MGRSSN} DEPARTMENT)$
    $RESULT \longleftarrow \Pi_{FNAME,\ MINIT,\ DNAME}\ (TEMP)$
    The result is in Figure 6.11 (Figure 7.18 on e3)

- **RIGHT OUTER JOIN:** similar to LEFT OUTER JOIN, but keeps every tuple t in right relation R_2 in the resulting relation R_3.

  - Notation: $\bowtie\!\sqsubset$

- **FULL OUTER JOIN:** $\supset\!\bowtie\!\sqsubset$

6.4.4 The OUTER UNION Operation

- **OUTER UNION:** make union of two relations that are partially compatible.
  - The list of compatible attributes should include a key for both relations.
  - $R_3(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m, C_1, C_2, \ldots, C_p) \longleftarrow R_1(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ OUTER UNION $R_2(A_1, A_2, \ldots, A_n, C_1, C_2, \ldots, C_p)$
  - Tuples from R_1 and R_2 with the same key value are represented once in R_3
  - In R_3, fill "NULL" if necessary
  - For example, STUDENT(NAME, SSN, DEPT, ADVISOR) and FACULTY(NAME, SSN, DEPT, RANK)

    The resulting relation schema after OUTER UNION will be R_3(NAME, SSN, DEPT, ADVISOR, RANK)

6.5 Examples of Queries in Relational Algebra

- Retrieve the name and address of all employees who work for the 'Research' department.

  $RESEARCH\_DEPT \longleftarrow \sigma_{DNAME='Research'}(DEPARTMENT)$

  $RESEARCH\_EMPS \longleftarrow (RESEARCH\_DEPT \bowtie_{DNUMBER=DNO} (EMPLOYEE))$

  $RESULT \longleftarrow \pi_{FNAME,LNAME,ADDRESS}(RESEARCH\_EMPS)$

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

  $STAFFORD\_PROJS \longleftarrow \sigma_{PLOCATION='Stafford'}(PROJECT)$

  $CONTR\_DEPT \longleftarrow (STAFFORD\_PROJS \bowtie_{DNUM=DNUMBER} (DEPARTMENT))$

  $PROJ\_DEPT\_MGR \longleftarrow (CONTR\_DEPT \bowtie_{MGRSSN=SSN} (EMPLOYEE))$

  $RESULT \longleftarrow \pi_{PNUMBET,DNUM,LNAME,ADDRESS,BDATE}(PROJ\_DEPT\_MGR)$

- Find the names of employees who work on all the projects controlled by department number 5.

  $DEPT5\_PROJS(PNO) \longleftarrow \pi_{PNUMBER}(\sigma_{DNUM=5}(PROJECT))$

  $EMP_PROJ(SSN, PNO) \longleftarrow \pi_{ESSN,PNO}(WORKS\_ON)$

  $RESULT\_EMP\_SSNS \longleftarrow EMP\_PROJ \div DEPT\_PROJS$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(RESULT\_EMP\_SSNS * EMPLOYEE)$

- Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

  $SMITHS(ESSN) \longleftarrow \pi_{SSN}(\sigma_{LNAME='Smith'}(EMPLOYEE))$

  $SMITH\_WORKER\_PROJ \longleftarrow \pi_{PNO}(WORKS\_ON * SMITHS)$

  $MGRS \longleftarrow \pi_{LNAME,DNUMBER}(EMPLOYEE \bowtie_{SSN=MGRSSN} DEPARTMENT)$

  $SMITH\_MANAGED\_DEPTS(DNUM) \longleftarrow \pi_{DNUMBER}(\sigma_{LNAME='Smith'}(MGRS))$

  $SMITH\_MGR\_PROJS(PNO) \longleftarrow \pi_{PNUMBER}(SMITH\_MANAGED\_DEPTS * PROJECT)$

  $RESULT \longleftarrow (SMITH\_WORKER\_PROJS \cup SMITH\_MGR\_PROJS)$

- List the names of all employees who have two or more dependents.

  $T_1(SSN, NO\_OF\_DEPTS) \longleftarrow_E SSN\Im_{COUNTDEPENT_NAME}(DEPENDENT)$

  $T_2 \longleftarrow \sigma_{NO\_OF\_DEPTS \geq 2}(T_1)$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(T_2 * EMPLOYEE)$

- Retrieve the names of employees who have no dependents.

  $ALL\_EMPS \longleftarrow \pi_{SSN}(EMPLOYEE)$

  $EMPS\_WITH\_DEPS(SSN) \longleftarrow \pi_{ESSN}(DEPENDENT)$

  $EMPS\_WITHOUT\_DEPS \longleftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$

- List the names of managers who have at least one dependent.

  $MGRS(SSN) \longleftarrow \pi_{MGRSSN}(DEPARTMENT)$

  $EMPS\_WITH\_DEPS(SSN) \longleftarrow \pi_{ESSN}(DEPENDENT)$

  $MGRS\_WITH\_DEPS \longleftarrow (MGRS \cap EMPS\_WITH\_DEPS)$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(MGRS\_WITH\_DEPS * EMPLOYEE)$

- Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

  $EMPS\_DEPS \longleftarrow$

  $(EMPLOYEE \bowtie_{SSN=ESSN \text{ AND } SEX=SEX \text{ AND } FNAME=DEPENDENT\_NAME} DEPENDENT)$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(EMPS\_DEPS)$

- Retrieve the names of all employees who do not have supervisors.

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(\sigma_{SUPERSSN=NULL}(EMPLOYEE))$

- Find the sum of salary of all employees, the maximum salary, the minimum salary, and the average salary for each department.

  $RESULT \longleftarrow$

  $_{DNO}\Im_{SUM \text{ } SALARY,\ MAXIMUM \text{ } SALARY,\ MINIMUM \text{ } SALARY,\ AVERAGE \text{ } SALARY}(EMPLOYEE)$

6.6 The Tuple Relational Calculus

- **Nonprocedural** Language: Specify what to do; Tuple (Relational) Calculus, Domain (Relational) Calculus.
- **Procedural** Language: Specify how to do; Relational Algebra.
- The expressive power of Relational Calculus and Relational Algebra is identical.
- A relational query language L is considered **relationally complete** if we can express in L any query that can be expressed in Relational Calculus.
- Most relational query language is relationally complete but have more expressive power than relational calculus (algebra) because of additional operations such as aggregate functions, grouping, and ordering.

6.6.1 Tuple Variables and Range Relations

- A **tuple variable** usually **range over** a particular database relation: the variable may take as its value any individual tuple from that relation.
- General Form: $\{t \mid COND(t)\}$
- Examples:
  - Find all employees whose salary is above \$50,000.
    $\{t \mid EMPLOYEE(t)\ and\ t.SALARY > 50000\}$
  - Find the first and last names of all employees whose salary is above \$50,000.
    $\{t.FNAME, t.LNAME \mid EMPLOYEE(t)\ and\ t.SALARY > 50000\}$
    Compare to:

    SELECT T.FNAME, T.LNAME
    FROM EMPLOYEE **AS** T
    WHERE T.SALARY > 50000;
- Three information should be specified in a tuple calculus expression.

- For each tuple variable t, the **range relation** R of t is specified as R(t). (FROM clause in SQL)
- A condition to select particular combinations of tuples. (WHERE clause in SQL)
- A set of attributes to be retrieved, the **requested attributes.** (SELECT clause in SQL)

- Example: Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.
  $\{t.BDATE, t.ADDRESS \mid EMPLOYEE(t)\, and\, t.FNAME =' John'\, and\, t.MINIT =' B'\, and\, t.LNAME =' Smith'\}$

6.6.2 Expressions and Formulas in Tuple Relation Calculus

- A general expression form:
  $\{t_1.A_1, t_2.A_2, \ldots, t_n.A_n \mid COND(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$
  Where $t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula**

- A **formula** is made up of one or more atoms, which can be one of the following.
  - An atom of the form $R(t_i)$ defines the range of the tuple variable t_i as the relation R.
    If the tuple variable t_i is assigned a tuple that is a member of R, then the atom is TRUE.
  - An atom of the form $t_i.A$ **op** $t_j.B$, where **op** is one of the comparison operators $\{=, >, \geq, <, \leq, \neq\}$.
    If the tuple variables t_i and t_j are assigned to tuples such that the values of the attributes $t_i.A$ and $t_j.B$ satisfy the condition, then the atom is TRUE.
  - An atom of the form $t_i.A$ **op** c or c **op** $t_j.B$.
    If the tuple variables t_i (or t_j) is assigned to a tuple such that the value of the attribute $t_i.A$ (or $t_j.B$) satisfies the condition, then the atom is TRUE.

- A **formula** is made up one or more atoms connected via the logical operators **and, or, not** and is defined recursively as follows.
  - Every atom is a formula.
  - If F_1 and F_2 are formulas, then so are $(F_1\ and\ F_2)$, $(F_1\ or\ F_2)$, $not(F_1)$, $not(F_2)$. And
    * $(F_1\ and\ F_2)$ is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
    * $(F_1\ or\ F_2)$ is FALSE if both F_1 and F_2 are FALSE; otherwise, it is TRUE.
    * $not(F_1)$ is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.
    * $not(F_2)$ is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

6.6.3 The Existential and Universal Quantifiers

- There are two **quantifiers** can appear in formula, **universal quantifier** $\forall$ and **existential quantifier** $\exists$.
- **free** and **bound** for tuple variables in formula.
  - An occurrence of a tuple variable in a formula F that is an atom is free in F.
  - An occurrence of a tuple variable t is free or bound in $(F_1\ and\ F_2)$, $(F_1\ or\ F_2)$, $not(F_1)$, $not(F_2)$, depending on whether it is free or bound in F_1 or F_2. Notice that a tuple variable may be free in F_1 and bound in F_2.
  - All free occurrences of a tuple variable t in F are bound in formulas $F' = (\exists\, t)(F)$ or $F' = (\forall\, t)(F)$. For example:
    $F_1 : d.DNAME =' Research'$
    $F_2 : (\exists\, t)(d.DNUMBER = t.DNO)$
    $F_3 : (\forall\, d)(d.MGRSSN =' 333445555')$
    Where variable d is free in F_1 and F_2, but bound in F_3. Variable t is bound in F_2.
- A formula with **quantifiers** is defined as follows.
  - If F is a formula, then so is $(\exists\, t)(F)$, where t is a tuple variable. The formula $(\exists\, t)(F)$ is TRUE if the formula F evaluates to TRUE for some tuple assigned to free occurrences of t in F; otherwise $(\exists\, t)(F)$ is FALSE.

– If F is a formula, then so is $(\forall t)(F)$, where t is a tuple variable. The formula $(\forall t)(F)$ is TRUE if the formula F evaluates to TRUE for every tuple (in the universe) assigned to free occurrences of t in F; otherwise $(\forall t)(F)$ is FALSE.

6.6.4 Example Queries Using the Existential Quantifier

- Retrieve the name and address of all employees who work for the 'Research' department.
  $\{t.FNAME, t.LNAME, t.ADDRESS \mid EMPLOYEE(t)\ and\ (\exists\, d)$
  $(DEPARTMENT(d)\ and\ d.DNAME =' Research'\ and\ d.DNUMBER = t.DNO)\}$

- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.
  $\{p.PNUMBER, p.DNUM, m.LNAME, m.BDATE, m.ADDRESS \mid PROJECT(p)$
  $and\ EMPLOYEE(m)\ and\ p.PLOCATION =' Stafford'\ and$
  $((\exists\, d)(DEPARTMENT(d)\ and\ p.DNUM = d.DNUMBER\ and\ d.MGRSSN = m.SSN))\}$

- For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.
  $\{e.FNAME, e.LNAME, s.FNAME, s.LNAME \mid EMPLOYEE(e)\ and$
  $EMPLOYEE(s)\ and\ e.SUPERSSN = s.SSN\}$

- Find the name of each employee who works on some project controlled by department number 5.
  $\{e.LNAME, e.FNAME \mid EMPLOYEE(e)\ and\ ((\exists\, x)(\exists\, w)$
  $(PROJECT(x)\ and\ WORKS\_ON(w)\ and\ x.DNUM = 5\ and\ w.ESSN = e.SSN\ and$
  $x.PNUMBER = w.PNO))\}$

- Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as manager of the controlling department for the project.
  $\{p.PNUMBER \mid PROJECT(p)\ and$
  $(((\exists\, e)(\exists\, w)(EMPLOYEE(e)\ and\ WORKS\_ON(w)\ and$

$w.PNO = p.PNUMBER$ *and* $e.LNAME =' Smith'$ *and* $e.SSN = w.ESSN))$

or

$((\exists m)(\exists d)(EMPLOYEE(m)$ *and* $DEPARTMENT(d)$ *and*

$p.DNUM = d.DNUMBER$ *and* $d.MGRSSN = m.SSN$ *and* $m.LNAME =' Smith')))\}$

6.6.5 Transforming the Universal and Existential Quantifiers

- $(\forall x)(P(x)) \equiv not(\exists x)(not(P(x)))$
- $(\exists x)(P(x)) \equiv not(\forall x)(not(P(x)))$
- $(\forall x)(P(x)$ *and* $Q(x)) \equiv not(\exists x)(not(P(x))$ *or* $not(Q(x)))$
- $(\forall x)(P(x)$ *or* $Q(x)) \equiv not(\exists x)(not(P(x))$ *and* $not(Q(x)))$
- $(\exists x)(P(x)$ *or* $Q(x)) \equiv not(\forall x)(not(P(x))$ *and* $not(Q(x)))$
- $(\exists x)(P(x)$ *and* $Q(x)) \equiv not(\forall x)(not(P(x))$ *or* $not(Q(x)))$
- $(\forall x)(P(x)) \Rightarrow (\exists x)(P(x))$
- $not(\exists x)(P(x)) \Rightarrow not(\forall x)(P(x))$

6.6.6 Using the Universal Quantifier

- Find the names of employees who work on all the projects controlled by department number 5.

  $\{e.LNAME, e.FNAME \mid EMPLOYEE(e)$ *and* $((\forall x)(not(PROJECT(x)$ *or*

  $not(x.DNUM = 5)$ *or* $((\exists w)(WORKS\_ON(w)$ *and* $w.ESSN = e.SSN$ *and*

  $x.PNUMBER = w.PNO))))\}$

  BREAK INTO:

  $\{e..LNAME, e.FNAME \mid EMPLOYEE(e)$ *and* $F'\}$

  $F' = ((\forall x)(not(PROJECT(x))$ *or* $F_1))$

  $F_1 = not(x.DNUM = 5)$ *or* F_2

$F_2 = ((\exists\, w)(WORKS\_ON(w)\ and\ w.ESSN = e.SSN\ and\ x.PNUMBER = w.PNO))$

IS EQUIVALENT TO:

$\{e.LNAME, e.FNAME \mid EMPLOYEE(e)\ and\ (not(\exists\, x)(PROJECT(x)$

$and\ (x.DNUM = 5)\ and$

$(not(\exists\, w)(WORKS\_ON(w)\ and\ w.ESSN = e.SSN\ and$

$x.PNUMBER = w.PNO))))\}$

- Find the names of employees who have no dependents.

  $\{e.FNAME, e.LNAME \mid EMPLOYEE(e)\ and\ (not(\exists\, d)(DEPENDENT(d)$

  $and\ e.SSN = d.ESSN))$

  IS EQUIVALENT TO:

  $\{e.FNAME, e.LNAME \mid EMPLOYEE(e)\ and\ ((\forall\, d)$

  $(not(DEPENDENT(d))\ or\ not(e.SSN = d.ESSN)))\}$

- List the names of managers who have at least one dependent.

  $\{e.FNAME, e.LNAME \mid EMPLOYEE(e)\ and\ ((\exists\, d)(\exists p)$

  $(DEPARTMENT(d)\ and\ DEPENDENT(p)\ ande.SSN = d.MGRSSN\ and$

  $p.ESSN = e.SSN))\}$

6.6.7 Safe Expressions

- **Safe Expression:** The result is a finite number of tuples.
- For example, $\{t \mid not(EMPLOYEE(t))\}$ is unsafe.
- **Domain of a tuple relational calculus expression:** The set of all values that either appear as constant values in the expression or exist in any tuple of the relations referenced in the expression.
- An expression is **safe** if all values in its result are from the domain of the expression.

6.7 The Domain Relational Calculus

- Rather than having variables range over tuples in relations, the **domain variables** range over single values from domains of attributes,

- General form: $\{x_1, x_2, \ldots, x_n \mid COND(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$
  Domain Variables: $x_1, x_2, \ldots, x_n$ that range over the domains of attributes.
  Formula: $COND$ is the formula or condition of the domain relational calculus.
  A formula is made up of **atoms**.

  - An atom of the form $R(x_1, x_2, \ldots, x_j)$ (or simply $R(x_1 x_2 \ldots x_n)$), where R is the name of a relation of degree j and each x_i, $1 \leq i \leq j$, is a domain variable.
    This atom defines that $< x_1, x_2, \ldots, x_j >$ must be a tuple in R, where the value of x_i is the value of the i^{th} attribute of the tuple.
    If the domain variables $x_1, x_2, \ldots, x_j$ are assigned values corresponding to a tuple of R, then the atom is TRUE.

  - An atom of the form x_i **op** x_j, where **op** is one of the comparison operators $\{=, >, \leq, <, \geq, \neq\}$.
    If the domain variables x_i and x_j are assigned values that satisfy the condition, then the atom is TRUE.

  - An atom of the form x_i **op** c or c **op** x_j, where c is a constant value.
    If the domain variables x_i (or x_j) is assigned a value that satisfies the condition, then the atom is TRUE.

- Examples: we use lowercase letters $l, m, n, \ldots, x, y, z$ for domain variables.

  - Retrieve the birthdate and address of the employee whose name is 'John B Smith'.
    $\{uv \mid (\exists q)(\exists r)(\exists s)(\exists t)(\exists w)(\exists x)(\exists y)(\exists z)$
    $(EMPLOYEE(qrstuvwxyz)\ and\ q =' John'\ and\ r =' B'\ and\ s =' Smith')\}$

    An alternative notation for this query.
    $\{uv \mid EMPLOYEE('John',' B',' Smith', t, u, v, w, x, y, z)\}$

    For convenience, we quantify only those variables actually appearing in a condition (these would be q, r and s in the above example) in the

rest of examples

- Retrieve the name and address of all employees who work for the 'Research' department.
  $\{qsv \mid (\exists z)(\exists l)(\exists m)(EMPLOYEE(qrstuvwxyz)\ and$
  $DEPARTMENT(lmno)\ and\ l =' Research'\ and\ m = z)\}$

- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.
  $\{iksuv \mid (\exists j)(\exists m)(\exists n)(\exists t)(PROJECT(hijk)\ and\ EMPLOYEE(qrstuvwxyz)$
  $and\ DEPARTMENT(lmno)\ and\ k = m\ and\ n = t\ and\ j =' Stafford')\}$

- Find the names of employees who have no dependents.
  $\{qs \mid (\exists t)(EMPLOYEE(qrstuvwxyz)\ and\ (not(\exists l)(DEPENDENT(lmnop)$
  $and\ t = l)))\}$

  IS EQUIVALENT TO:
  $\{qs \mid (\exists t)(EMPLOYEE(qrstuvwxyz)\ and\ ((\forall l)(not(DEPENDENT(lmnop))$
  $or\ not(t = l))))\}$

- List the names of managers who have at least one dependent.
  $\{sq \mid (\exists t)(\exists j)(\exists l)(EMPLOYEE(qrstuvwxyz)\ and\ DEPARTMENT(hijk)$
  $and\ DEPENDENT(lmnop)\ and\ t = j\ and\ l = t)\}$

Chapter 7, ER- and EER-to-Relational Mapping, and Other Relational Languages

7.1 Relational Database Design Using ER-to-Relational Mapping

7.1.1 ER-to-Relation Mapping Algorithm

Compare Figure 7.1 and 7.2 (Figure 3.15 and 7.5 on e3).

- Step 1: Strong entity type $E \overset{CREATE}{\longrightarrow}$
  - A relation R contains all simple and simple component attributes of E;
  - Choose the key of E as the primary key of R
- Step 2: Weak entity type W with owner entity type $E \overset{CREATE}{\longrightarrow}$
  - A relation R contains all simple and all simple component attributes of W;
  - Add the primary key attributes of E and serve as a foreign key of R.
  - Combine the primary key of E and the partial key of W as an primary key of R.
- Step 3: Binary 1 : 1 relationship type R identifying S and T relations $\longrightarrow$
  - Choose one from S and T which has the total participation in R. (Because we want to avoid null values)
  - Assume we select S, then add all simple and simple component attributes of R to the relation S.
  - Add the primary key of T to S and serve as a foreign key of S.
- Step 4: Binary 1 : N relationship type R identifying S and T relations (suppose S is in the N side) $\longrightarrow$
  - Add all simple and simple component attributes of R to S;
  - Add the primary key of T to S and serve as a foreign key of S;

- If we do not choose N side, it will violate key constraint.

- Step 5: Binary M : N relationship type R identifying S and T relations $\overset{CREATE}{\longrightarrow}$
  - A relation U contains all simple and simple component attributes of R;
  - Add the primary key of S to U and serve as a foreign key of U;
  - Add the primary key of T to U and serve as a foreign key of U;
  - The combination of two foreign keys forms the primary key of U;

- Step 6: Multivalued attribute A $\overset{CREATE}{\longrightarrow}$
  - A relation R contains A;
  - Add the primary key k - serve as a foreign key - of the relation that represents the entity type or relationship type that has A as an attribute;
  - The combination of A and k is the primary key of R.

- Step 7: n-ary relationship type R (where n > 2) $\overset{CREATE}{\longrightarrow}$
  - A relation S contains all simple and simple component attributes of R;
  - Add the primary keys of all participating relations to S and serve as foreign keys of S.
  - The primary key of S: the combination of all foreign keys except those referencing relations with cardinality ratio 1 on R.

7.2 Mapping EER Model Constructs to Relations

7.2.1 Mapping of Specialization or Generalization

- Step 8: m subclasses $\{S_1, S_2, \ldots, S_m\}$ and superclass C with $Attr(C) = \{k, a_1, a_2, \ldots, a_n\}$, where k is the key of C.
  - Option 8A: Multiple relations – Superclass and subclasses
    - For the superclass C, create a relation $L(k, a_1, a_2, \ldots, a_n)$ and $PK(L) = k$.

    - For each subclass S_i, create a relation $L_i(ATTR(S_i) \cup \{k\})$ with $PK(L_i) = k$.
    - This option works for total/partial, disjoint/overlapping.
  - Option 8B: Multiple relations – Subclass relations only
    - For each subclass S_i, create a relation $L_i(ATTR(S_i) \cup \{k, a_1, a_2, \ldots, a_n\})$ and $PK(L_i) = k$.
    - This option is for total participation and disjoint.
  - Option 8C: Single relation with one type attribute
    - For the superclass C, create a relation $L(\{k, a_1, a_2, \ldots, a_n\} \cup ATTR(S_1) \cup ATTR(S_2) \cup \ldots \cup ATTR(S_m) \cup \{t\})$ and $PK(L) = k$, where t is a type attribute indicating which subclass each tuple belongs to.
    - This option is for disjoint.
  - Option 8D: Single relation with multiple type attributes
    - For the superclass C, create a relation $L(\{k, a_1, a_2, \ldots, a_n\} \cup ATTR(S_1) \cup ATTR(S_2) \cup \ldots \cup ATTR(S_m) \cup \{t_1, t_2, \ldots, t_m\})$ and $PK(L) = k$, where each t_i is a boolean attribute indicating whether a tuple belongs to the subclass S_i.
    - This option is for overlapping (also disjoint).

- Summary:

Options	Works for	Disadvantage
8A	Total/Partial; Disjoint/Overlapping	Need QUUIJOIN to retrieve the special and inherited attributes of entities in S_i
8B	Total; Disjoint	Need OUTER UNION to retrieve all entities in superclass C
8C	Total/Partial; Disjoint	Lots of NULL values
8D	Total/Partial; Disjoint/Overlapping	Lots of NULL values

- Example, Figure 7.4 (Figure 9.2 on e3) is mapping from the EER schema in Figure 4.4 (Figure 4.4 on e3).

7.2.2 Mapping of Shared Subclasses

- A shared subclass is a subclass of several superclasses. These classes must have the same key attribute; otherwise, the shared subclass would be modeled as a category.
- Any options in Step 8 can be used to a shared subclass, although usually option 8A is used. See Figure 7.5 (Fig 9.3 on e3).

7.2.3 Mapping of Category

- One category C and m superclasses $\{S_1, S_2, \ldots, S_m\}$.
  - Superclasses with different keys: let k_{S_i} denote the key of S_i.
    - For the category C, create a relation $L(ATTR(C) \cup \{a\ surrogate\ key\ k_r\})$ and $PK(L) = K_r$.
    - For each superclasses S_i, create a relation $L_i(ATTR(S_i) \cup \{k_r\})$ and $PK(L_i) = k_{S_i}$ and $FK(L_i) = k_r$ referencing relation L.
  - Superclasses with the same key k_s:
    - For the category C, create a relation $L(ATTR(S_i) \cup \{k_s\})$ and $PK(L) = k_s$.
    - For each superclasses S_i, create a relation $L_i(ATTR(S_i))$ and $PK(L_i) = k_s$.
    - Example, Figure 7.6 (Figure 9.4 on e3) is mapping from Figure 4.7 (Figure 4.7 on e3).