Chapter 3, Data Modeling Using the Entity-Relationship Model

• A Company Database Application Example

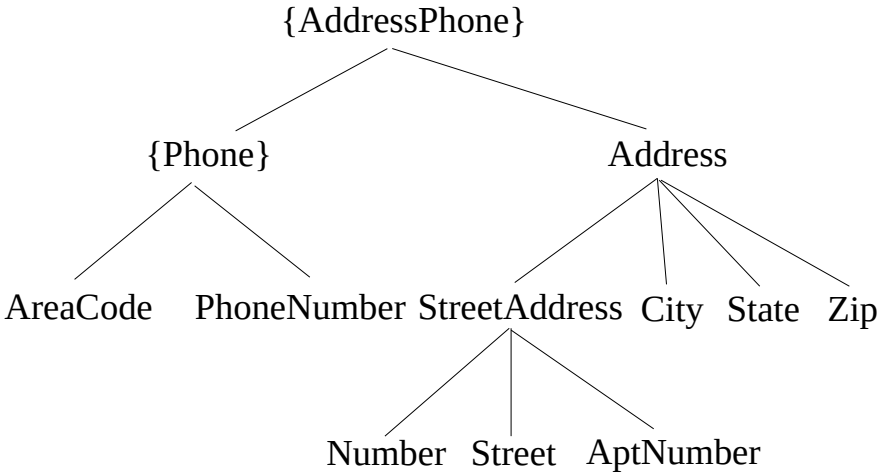
After the requirements collection and analysis phase, the database designers stated the following description of the Company

- 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 ⟶ 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

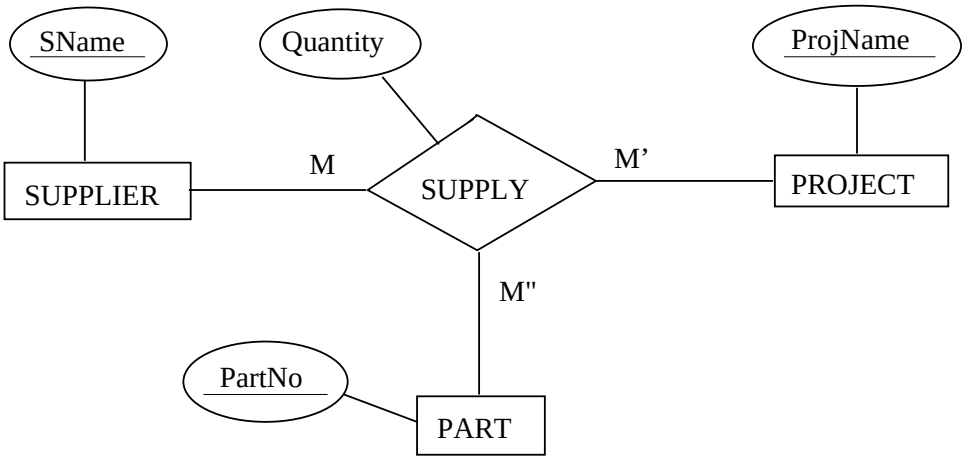
New R Types	Participating E Types	Ratio	Participation	Attributes
MANAGES	DEPARTMENT, EMPLOYEE	1:1	Total, Partial	StartDate
WORKS_FOR	DEPARTMENT, EMPLOYEE	1:N	Total, Total	
CONTROLS	DEPARTMENT, PROJECT	1:N	Partial, Total	
SUPERVISION	EMPLOYEE, EMPLOYEE	1:N	Partial, Partial	
WORKS_ON	EMPLOYEE, PROJECT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

- **Relationship Types of Degree Higher than Two:**
  - 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.

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

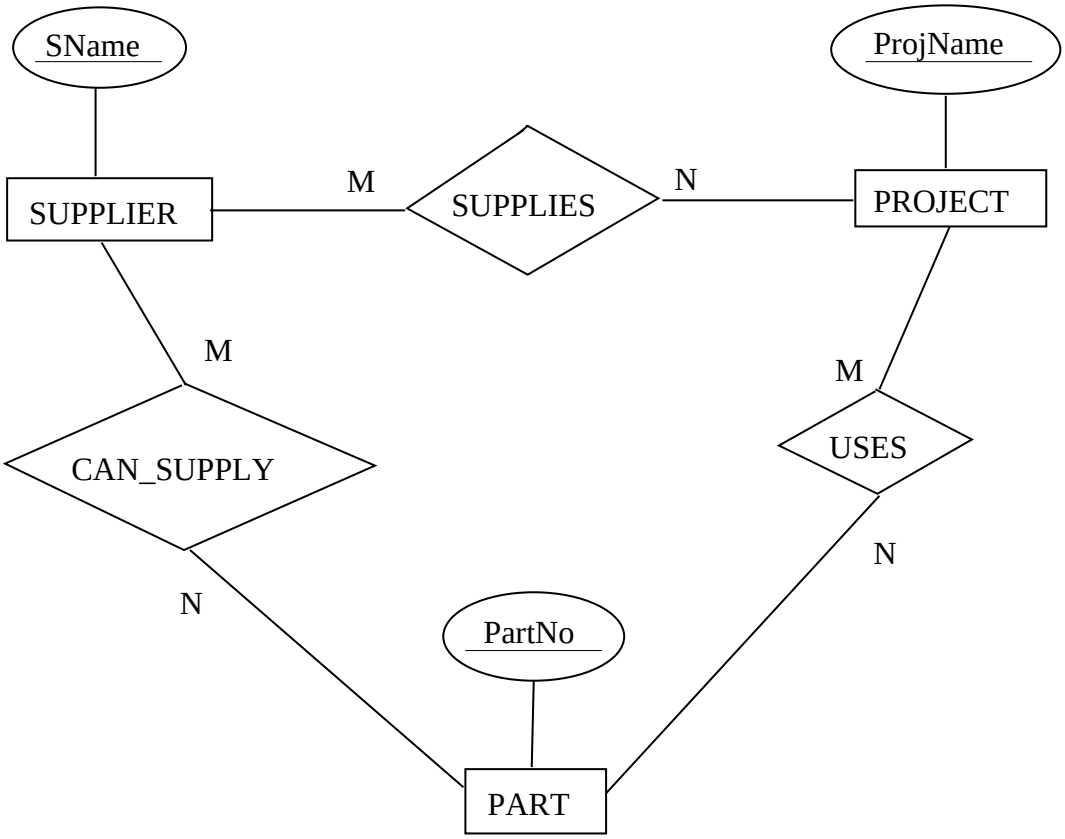
$$(s_2, j_1, p_1), (s_2, j_3, p_1), (s_2, j_2, p_3), (s_2, j_3, p_3)$$

      For a (s_x, j_y) combination, we may have multiple p_z
      For a (j_y, p_z) combination, we may have multiple s_x
      For a (s_x, p_z) combination, we may 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 any 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) & $(s,j) \not\Longrightarrow (s,j,p)$, but
  $(s,j,p) \Longrightarrow (s,p)$ & (j,p) & (s,j)
* 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_1,p_1,j_1) may not exist. It could be (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 3.17(c) (Fig 4.13(c) on e3).

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 DateOfVote 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.

Exercise 3.22

A database is being constructed to keep track of the teams and games of a baseball league. A team has a number of players, not all of whom participate in each game. It is desired to keep track of the players participating in each game for each team, the positions they played in that game, and the result of the game. Design an ER schema diagram for this application, stating any assumptions 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 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 a 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:** 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.

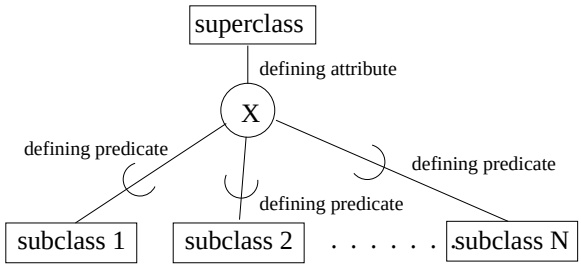
- The inheritance of a category vs. the inheritance of a shared subclass.
  OWNER in Figure 4.8 (Fig 4.8 on e3) vs. ENGINEERING_MANAGER in Figure 4.6 (Fig 4.6 on e3).

- 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).

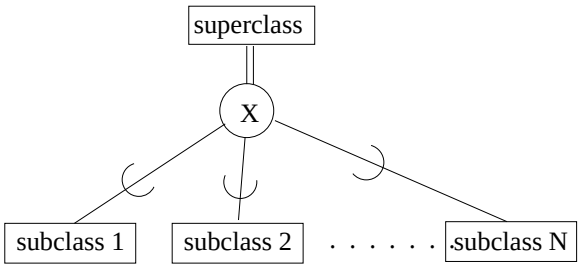
- A category can be **total** or **partial**. See Figure 4.9 on e3.

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

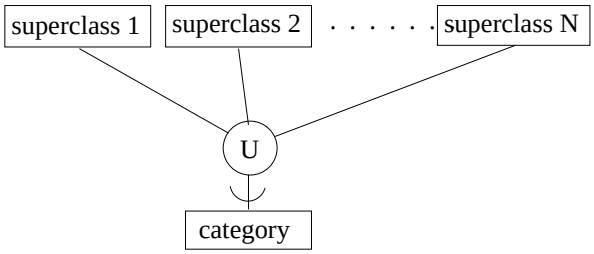
- **The Notation for EER Model:** See next page.



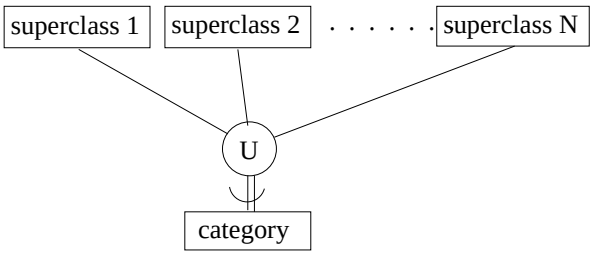
1. Partial Specialization/Generalization
2. X = ’d’ means Disjoint
3. X = ’o’ means overlap
4. The subset sign indicates the direction of a C/S relationship



Total Specialization/Generalization



1. Partial Category
2. U means UNION



Total Category

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 **atomic** (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 $t[A_i]$ must be atomic and belong to $dom(A_i) \cup NULL$.

5.2.2 Key Constraints and Constraints on NULL

- A **superkey** SK of a schema R 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 $FK = \{A_u, A_v, \ldots, A_z\}$ 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, $\exists$ a 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. 162).
  - 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 : $< Clause >< Boolean\ OP >< Clause >< Boolean\ OP > \ldots$
  - Clause : $< Attribute\ Name >< Comparison\ OP >< Const\ Value >$ or $< Attribute\ Name >< Comparison\ OP >< Attribute\ Name >$
  - See examples on page 163 of the book.
  - Deletion may violate referential integrity constraint.
  - If there is a violation, three options: rejection, cascade the deletion, or modify the referencing attribute values.

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.
- See examples on page 164 on the book.

Exercise 5.11 on page 166 of the book.

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

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

    $$|\ R_2\ | \leq |\ R_1\ |\quad for\ any\ selection\ condition$$
  - 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.
  - $R_2 \leftarrow \pi_{<attribute\ list>}(R_1)$
  - $R_2 \leftarrow \pi_{LNAME,\ SALARY}(EMPLOYEE)$
  - 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 $\mid R_2 \mid = \mid R_1 \mid$
- $\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$, where
  $< list2 >$ must be a subset of $< list1 >$.

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,

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}

  - See Figure 6.5 (Fig 7.12 on e3). This figures shows a possible sequence of steps to retrieve a list of names of each female employee's dependents.

6.3 Binary 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),

    $$\text{ACTUAL\_DEPENDENT} \longleftarrow \text{EMPLOYEE} \bowtie_{SSN=ESSN} \text{DEPENDENT}$$
    $$\equiv \text{ACTUAL\_DEPENDENT} \longleftarrow \sigma_{SSN=ESSN}\ (\text{EMPLOYEE} \times \text{DEPENDENT})$$
- 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 6.7 (Figure 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) $\longleftarrow$ π_{SSN} ($\sigma_{FNAME='John'\ AND\ LNAME='Smith'}$ (EMPLOYEE))
  JSMITH_PROJ $\longleftarrow$ π_{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**.

  - Each group results in a tuple in R_2.

  - 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:** $R_3(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m) \longleftarrow R_1(A_1, A_2, \ldots, A_n) \ \text{⟕}_{<JOIN\ COND.>}\ R_2(B_1, B_2, \ldots, B_m)$

  - 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,

    $TEMP \longleftarrow (EMPLOYEE \ \text{⟕}_{SSN=MGRSSN}\ DEPARTMENT)$

    $RESULT \longleftarrow \pi_{FNAME,\ MINIT,\ DNAME}\ (TEMP)$

    The result is in Figure 6.12 (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: ⟖

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

6.4.4 The OUTER UNION Operation

- **OUTER UNION:** make union of two relations that are partially compatible.
  - $R_3(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m, C_1, C_2, \ldots, C_p) \longleftarrow R_1(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ OUTER UNION $R_2(A_1, A_2, \ldots, A_n, C_1, C_2, \ldots, C_p)$
  - The list of compatible attributes are represented only once in R_3.
  - Tuples from R_1 and R_2 with the same values on the set of compitable attributes are represented only 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.
- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.
- Find the names of employees who work on all the projects controlled by department number 5.
- 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.
- List the names of all employees who have two or more dependents.
- Retrieve the names of employees who have no dependents.
- List the names of managers who have at least one dependent.
- Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.
- Retrieve the names of all employees who do not have supervisors.
- Find the sum of salary of all employees, the maximum salary, the minimum salary, and the average salary for each department.

- 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.

$T_1(SSN, NO\_OF\_DEPTS) \longleftarrow_{ESSN} \Im_{COUNT\ DEPENDENT\_NAME}(DEPENDENT)$

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

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

- 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.

  $EMPS\_DEPS \longleftarrow$

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

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

- 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.

  $RESULT \longleftarrow$

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

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 a 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.

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

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

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

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 $R(x_1 x_2 \ldots x_j)$), 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.
    - For example, Figure 7.5(a) (Fig 9.2(a) on e3) is mapping from Figure 4.4 (Fig 4.4 on e3).
  - 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.
    - For example, Figure 7.5(b) (Fig 9.2(b) on e3) is mapping from Figure 4.3(b) (Fig 4.3(b) on e3).
  - 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 may generate some NULL values.
    - This option is for disjoint.
    - For example, Figure 7.5(c) (Fig 9.2(c) on e3) is mapping from Figure 4.4 (Fig 4.4 on e3).
  - 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).
    - For example, Figure 7.5(d) (Fig 9.2(d) on e3) is mapping from Figure 4.5 (Fig 4.5 on e3).

- Summary:

Options	Works for	Disadvantage
8A	Total/Partial; Disjoint/Overlapping	Need EQUIJOIN 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

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 7.6 (Fig 9.3 on e3) is mapping from Figure 4.7 (Fig 4.7 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 $L(ATTR(C) \cup \{k_s\})$ 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 7.7 (Figure 9.4 on e3) is mapping from Figure 4.8 (Figure 4.8 on e3).

Chapter 8, SQL-99: Schema Definition, Basic Constraints, and Queriezs

8.1 SQL Data Definition and Data Types

8.1.1 Schema and Catalog Concepts in SQL

- Schema in SQL: A schema name, authorization ID (who owns the schema), descriptors of each element - tables, constraints, views, domains and other constructs describing the schema.
  - A schema can be assigned a name and authorization ID, and the elements can be defined later.
  - For example, the following statement creates a schema called COMPANY, owned by the user with authorization ID 'JSMITH':

    CREATE SCHEMA COMPANY **AUTHORIZATION** JSMITH;

- Catalog in SQL: A named collection of schemas.
  - INFORMATION-SCHEMA in catalog – provides all element descriptors of all schemas to authorized users.
  - Referential integrity constraints can be defined between two relations within the same catalog.
  - Schemas within the same catalog can share certain elements.

8.1.2 The CREATE TABLE Command in SQL

- Syntax of CREATE TABLE command: refer to Table 8.2 (Table 8.1 on e3) on pp 245 for syntax; Figure 8.1, 8.2 (Fig 8.1(a), (b) on e3) for example.

8.1.3 Attribute Data Types and Domains in SQL

- Build-in data types:

- Numeric
    - INTEGER / INT, SMALLINT
    - FLOAT, REAL, DOUBLE PRECISION
    - DECIMAL(i,j) / DEC(i,j) / NUMERIC(i,j)

      where i: total # of decimal digits

      j: total # of digits after decimal point. (default 0)
- Character-string

  Fixed length – CHAR(n) / CHARACTER(n) – length n

  Varying length – VARCHAR(n) / CHAR VARYING(n) /CHARACTER VARYING(n) – length up to n.
- Bit-string

  Fixed length – BIT(n)

  Varying length – BIT VARYING(n) – length up to n.

  Note that the default of n in Character-string and Bit-string is 1 $\rightarrow$ CHAR and BIT mean length 1.
- DATE – YYYY-MM-DD (10 positions)
- TIME – HH : MM : SS (8 position)
- TIME(i) – HH : MM : SS : i positions for decimal fractions of a second
- TIME WITH TIME ZONE – HH : MM : SS_ _ _ _ _ _ (The six positions could be in the range +13:00 to -12:59).
- TIMESTAMP – (DATE, TIME, at least 6 positions for decimal fractions of a second, optional WITH TIME ZONE qualifier)
- INTERVAL – a relative value to increment or decrement an absolute value of a DATE, TIME, TIMESTAMP. Interval could be YEAR/MONTH or DAY/TIME.

- Define your own Data Types – Domain Declaration.
    - **CREATE DOMAIN** BDATE-TYPE **AS** CHAR(8);

- **CREATE DOMAIN** SSN-TYPE **AS** CHAR(9);

8.2 Specifying Basic Constraints in SQL

8.2.1 Specifying Attribute Constraints and Attribute Defaults

- Attribute constraints:
  - **NOT NULL**: This attribute could not have null value. Primary key attribute should always be NOT NULL.
  - **DEFAULTL** v : Any new tuple without specifying a value for this attribute, the default value v should be used.
- Use **CHECK** clause to restrict an attribute or a domain values.
  DNUMBER INT **NOT NULL CHECK** (DNUMBER $>$ 0 **AND** DNUMBER $<$ 21);
  CREATE DOMAIN D_NUM **AS** INTEGER **CHECK** (D_NUM $>$ 0 **AND** D_NUM $<$ 21);

8.2.2 Specifying Key and Referential Integrity Constraints

- Keys constraint:
  - [CONSTRAINT C_Name]
    PRIMARY KEY (List of attributes as the primary key)
  - [CONSTRAINT C_Name]
    UNIQUE (List of attributes as the secondary key)
- Referential integrity constraint:

  [CONSTRAINT C_Name]
  FOREIGN KEY (List of attributes as the foreign key) **REFERENCES**
  Referenced_Relation_Name (primary key attribute list)
  [ON DELETE SET NULL / SET DEFAULT / CASCADE]
  [ON UPDATE SET NULL / SET DEFAULT / CASCADE]

- Example, see Figure 8.1 and 8.2 (Fig 8.1(a) and (b) on e3).

8.2.4 Specifying Constraints on Tuples Using CHECK

- Tuple-based constraints should be checked while inserting a new tuple to the table.
- Using **CHECK** clause at the end of the CREATE TABLE statement to specify this type of constraints.
  CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);

8.3 Schema Change Statements in SQL

8.3.1 The DROP Command

- **DROP SCHEMA** COMPANY **CASCADE**;
  - remove the company schema and all its elements.
- **DROP SCHEMA** COMPANY **RESTRICT**;
  - remove the company schema only if it has no elements in it.
- **DROP TABLE** DEPENDENT **CASCADE**;
  - remove the DEPENDENT table and all referencing constraints of other tables and views.
- **DROP TABLE** DEPENDENT **RESTRICT**;
  - remove the DEPENDENT table only if it is not referenced by other tables or views.

8.3.2 The ALTER Command

- Adding s column (attribute):
  - **ALTER TABLE** COMPANY.EMPLOYEE **ADD** JOB VARCHAR(12) [**NOT NULL**] [**DEFAULT** v];

- Dropping a column (attribute):
  - **ALTER TABLE** COMPANY.EMPLOYEE **DROP** ADDRESS **CASCADE**;
    - Drop EMPLOYEE.ADDRESS and all constraints and views that reference this column.
  - **ALTER TABLE** COMPANY.EMPLOYEE **DROP** ADDRESS **RESTRICT**;
    - Drop EMPLOYEE.ADDRESS only if no views or constraints reference this column.
- Changing a column definition:
  - **ALTER TABLE** DEPARTMENT **ALTER** MGRSSN **DROP DEFAULT**;
    DROP NOT NULL;
    SET DEFAULT v;
    SET NOT NULL;
- Adding a table constraint:
  - **ALTER TABLE** COMPANY.EMPLOYEE **ADD CONSTRAINT** (List of Constraints);
- Dropping a table constraint:
  - **ALTER TABLE** COMPANY.EMPLOYEE **DROP CONSTRAINT** C_Name **CASCADE**;
    - Drop this constraint from Employee and all other tables.
  - **ALTER TABLE** COMPANY.EMPLOYEE **DROP CONSTRAINT** C_Name **RESTRICT**;
    - Drop this constraint from EMPLOYEE only.

8.4 Basic Queries in SQL

SOL specify what to do for a query instead of how to do it. Let the DBMS to handle the detailed operations and optimization.

8.4.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

SQL uses **SELECT-FROM-WHERE block** to specify a query.

SELECT {**attribute list**}[1]
FROM {**table list**}[2]
WHERE {**condition**}[3];

Examples:

Q0: Retrieve the birthdate and address of the employee(s) whose name is 'John B. Smith'.

- Result is shown in Figure 8.3(a) (Fig 8.2(a) on e3)

SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME='John' **AND** MINIT='B' **AND**
LNAME='Smith';

Q1: Retrieve the name and address of all employees who work for the 'Research' department. Result is shown in Figure 8.3(b) (Fig 8.2(b) on e3)

SELECT FNAME, LNAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME='Research' **AND** DNUMBER=DNO;

Q2: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's name, address, and birthdate. Result is shown in Figure 8.3(c) (Fig 8.2(c) on e3)

SELECT PNUMBER, DNUM, LNAME, ADDRESS, BDATE
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUM=DNUMBER **AND** MGRSSN=SSN **AND**
PLOCATION='Stafford';

[1]List of attributes to be retrieved.
[2]List of tables required for the process.
[3]Conditions to identify tuples to be retrieved.

8.4.2 Ambiguous Attribute Names, Renaming (Aliasing), and Tuple Variables

Q8: For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

- Result is shown in Figure 8.3(d) (Fig 8.2(d) on e3)

```
SELECT    E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM      EMPLOYEE [AS][4] E, EMPLOYEE [AS] S
WHERE     E.SUPERSSN=S.SSN;
```

Here E and S are tuple variables.
Attributes can also be renamed within the query.
EMPLOYEE [**AS**] E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)

8.4.3 Unspecified WHERE-Clause and Use of Asterisk (*)

Select all EMPLOYEE SSNs **(Q9)**, and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME **(Q10)** in the database.

- Result is shown in Figure 8.3(e) and (f) (Fig 8.2(e), (f) on e3)

```
SELECT    SSN
FROM      EMPLOYEE[5];

SELECT    SSN, DNAME
FROM      EMPLOYEE, DEPARTMENT[6];
```

Q1C: Retrieve all the attribute values of EMPLOYEE who work in DEPARTMENT number 5. Result is shown in Figure 8.3(g) (Fig 8.2(g) on e3)

```
SELECT[7]    *
FROM         EMPLOYEE
WHERE        DNO=5;
```

[4] [ ] means optional.
[5] All tuples in EMPLOYEE are qualified.
[6] All tuples in the CROSS PRODUCT are qualified.

Q1D: Retrieve all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT he or she works in for every employee of the 'Research' department.

```
SELECT    *
FROM      EMPLOYEE, DEPARTMENT
WHERE     DNAME='Research' AND DNO=DNUMBER;
```

Q10A: Specify the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT.

```
SELECT    *
FROM      EMPLOYEE, DEPARTMENT;
```

8.4.4 Tables as Sets in SQL

SQL query returning table allows duplicate tuples. Use **DISTINCT** to eliminate duplicate tuples in the **SELECT** clause.

Retrieve the salary of every employee **(Q11)** and all distinct salary values **(Q11A)**. Result is shown in Figure 8.4(a) and (b) (Fig 8.3(a), (b) on e3)

```
SELECT [ALL]      SALARY
FROM              EMPLOYEE;

SELECT DISTINCT   SALARY
FROM              EMPLOYEE;
```

- As a set, the set operations are set union (**UNION**), set difference (**EXCEPT** or **MINUS**), and set intersection (**INTERSECT**) in SQL.
- These set operations return a set of tuples **without** duplication.

Q4: Make a list of all 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.

[7] * selects all attributes.

[8] [ ] means optional.

```
(SELECT DISTINCT  PNUMBER
FROM              EMPLOYEE, DEPARTMENT, PROJECT
WHERE             DNUM=DNUMBER AND SSN=MGRSSN
                  AND LNAME='Smith')
UNION[9]
(SELECT DISTINCT  PNO
FROM              EMPLOYEE, WORKS_ON
WHERE             SSN=ESSN AND LNAME='Smith');
```

Query: Please list the social security numbers of all managers who have no dependents.

```
(SELECT   MGRSSN
FROM      DEPARTMENT)
MINUS
(SELECT   ESSN
FROM      DEPENDENT);
```

How set operations work, see Figure 8.5.

8.4.5 Substring Pattern Matching and Arithmetic Operators

- '%' – replaces any number of zero or more characters.
- '_' – replaces a single character.
- If these two characters are possible literal characters in a string, use an escape character and a key word **ESCAPE**.
  For example, 'AB/_CD/%EF' **ESCAPE** '/'
  represents the literal string 'AB_CD%EF'

(Q12:) Retrieve all employee whose address is in Houston, Texas.

```
SELECT    FNAME, LNAME
FROM      EMPLOYEE
WHERE     ADDRESS LIKE '%Houston, Texas%';
```

[9]SQL also has EXCEPT and INTERSECT set operations.

(Q12A:) Retrieve all employee who were born during the 1950s

```
SELECT    FNAME, LNAME
FROM      EMPLOYEE
WHERE     BDATE LIKE '_ _ 5 _ - _ _ - _ _';
```

(Q13:) Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
SELECT    FNAME, LNAME, 1.1*SALARY[10]
FROM      EMPLOYEE, WORKS_ON, PROJECT
WHERE     SSN=ESSN AND PNO=PNUMBER AND
          PNAME='ProductX';
```

(Q14:) Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
SELECT    *
FROM      EMPLOYEE
WHERE     (SALARY BETWEEN 30000 AND 40000)[11]AND
          DNO=5;
```

8.4.6 Ordering of Query Results

(Q15:) Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.

```
SELECT    DNAME, LNAME, FNAME, PNAME
FROM      DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
WHERE     DNUMBER=DNO  AND SSN=ESSN AND
          PNO=PNUMBER
ORDER BY[12] DNAME, LNAME, FNAME
```

[10] Arithmetic operators for numeric values are + (addition), - (subtraction), * (multiplication), / (division). SQL also has a string concatenate operator ||, and '+' incrementing or '-' decrementing operators for time-related data types.

[11] it is equivalent to (SALARY $\geq$ 30000) **AND** (SALARY $\leq$ 40000)

8.5 More Complex SQL Queries

8.5.1 Comparisons involving NULL and Three-Valued Logic

- a **NULL** value may represent any one of three different meanings - unknown, not available, or not applicable.
- When a **NULL** value is involved in a comparison operation, the result is UNKNOWN. Therefore, three-valued logic used in SQL – TRUE, FALSE, UNKNOWN.
- Table 8.1 shows the results of three-valued logical expression if logical AND, OR, NOT are used.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
NOT			
TRUE	FALSE		
FALSE	TRUE		
UNKNOWN	UNKNOWN		

- Rather than using comparison operators to compare an attribute value to **NULL**, SQL uses **IS** or **IS NOT**.

Q18: Retrieve the names of all employees who do not have supervisors.
- Result is shown in Figure 8.4(d) (Fig 8.3(d) on e3)

[12]The default order is ascending, or you can put **ASC** or **DESC** keywords after the attributes to be ordered.

```
SELECT    FNAME, LNAME
FROM      EMPLOYEE
WHERE     SUPERSSN IS NULL;
```

8.5.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- Some queries may fetch existing values in the database, and then used in a comparison condition. Such queries can be formulated as **nested queries**.

Q4A: Make a list of all 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.

```
SELECT   DISTINCT PNUMBER
FROM     PROJECT
WHERE    PNUMBER IN   (SELECT  PNUMBER
                      FROM    PROJECT, DEPARTMENT
                              EMPLOYEE
                      WHERE   DNUM=DNUMBER AND
                              MGRSSN=SSN AND
                              LNAME='Smith')
         OR
         PNUMBER IN   (SELECT  PNO
                      FROM    WORKS_ON, EMPLOYEE
                      WHERE   ESSN=SSN AND
                              LNAME='Smith');
```

(Q29:) Retrieve the social security numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose SSN='123456789') works on.

```
SELECT   DISTINCT ESSN
FROM     WORKS_ON
WHERE    (PNO, HOURS) IN   (SELECT  PNO, HOURS
```

```
                    FROM    WORKS_ON
                    WHERE   ESSN='123456789');
```

In addition to **IN** operator, any combination of $\{=, >, <, \geq, \leq, <>\}$ and {**ANY, SOME, ALL**} can be used to compare a single value v to a set of values V.

(Q30:) Retrieve the names of employees whose salary is greater than the salary of all employees in department 5.

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > ALL (SELECT  SALARY
                      FROM    EMPLOYEE
                      WHERE   DNO=5);
```

Dealing the ambiguity of attribute names of nested queries.

(Q16:) Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

- Result is shown in Figure 8.4(c) (Fig 8.3(c) on e3)

```
SELECT  E.FNAME, E.LNAME
FROM    EMPLOYEE E
WHERE   E.SSN IN  (SELECT  ESSN
                   FROM    DEPENDENT
                   WHERE   E.FNAME=DEPENDENT_NAME
                           AND E.SEX=SEX);
```

8.5.3 Correlated Nested Queries

- If a inner query references to attributes declared in the outer query, the two queries are **correlated**.

- The better way to understand a nested query – the inner query is evaluated once for each tuple in the outer query.
- This kind of **correlated** queries can always be expressed as a single block query. We can rewrite **Q16** as follows.

```
SELECT   E.FNAME, E.LNAME
FROM     EMPLOYEE E, DEPENDENT D
WHERE    E.SSN=D.ESSN AND E.SEX=D.SEX AND
         E.FNAME=D.DEPENDENT_NAME;
```

Most commercial implementations of SQL do not have the set operator **CONTAINS** which return ture if one set contains all values of the other set.

(Q3:) Retrieve the name of each employee who works on all the projects controlled by department number 5.

```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    ( (SELECT  PNO
           FROM     WORKS_ON
           WHERE    ESSN=SSN)
           CONTAINS
           (SELECT  PNUMBER
           FROM     PROJECT
           WHERE    DNUM=5));
```

We use the combination of **NOT EXISTS** and **EXCEPT (MINUS)** functions to replace the **CONTAINS** operator.

8.5.4 The EXISTS, NOT EXISTS, and UNIQUE Functions in SQL

- EXISTS(Q) returns true if at least one tuple in the result of query Q.
- NOT EXISTS(Q) returns true if there are no tuples in the result of query Q.

- UNIQUE(Q) returns true if there are no duplicate tuples in the result of query Q.

(Q16:) Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

```
SELECT   E.FNAME, E.LNAME
FROM     EMPLOYEE E
WHERE    EXISTS  (SELECT  *
                  FROM    DEPENDENT
                  WHERE   E.FNAME=DEPENDENT_NAME
                          E.SEX=SEX AND E.SSN=ESSN);
```

(Q6:) Retrieve the names of employees who have no dependents.

```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    NOT EXISTS  (SELECT  *
                      FROM    DEPENDENT
                      WHERE   SSN=ESSN);
```

(Q7:) List the names of managers who have at least one dependent.

```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    EXISTS  (SELECT  *
                  FROM    DEPENDENT
                  WHERE   SSN=ESSN)
         AND
         EXISTS  (SELECT  *
                  FROM    DEPARTMENT
                  WHERE   SSN=MGRSSN);
```

Now we are ready to use **EXISTS** function to replace the **CONTAINS** set operator based on the set theory that (S1 CONTAINS S2) ≡ (S2 EXCEPT S1) is empty.

(Q3:) Retrieve the name of each employee who works on all the projects controlled by department number 5.

```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    NOT EXISTS
         ( (SELECT  PNUMBER
           FROM     PROJECT
           WHERE    DNUM=5)
           EXCEPT
           (SELECT  PNO
           FROM     WORKS_ON
           WHERE    SSN=ESSN));
```

We rephrase the query 3 as:
(Q3:) Retrieve the name of each employee such that there does not exist a project controlled by department 5 that the employee does not work on.

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   NOT EXISTS
        (SELECT *
        FROM    WORKS_ON B
        WHERE   (B.PNO IN       (SELECT     PNUMBER
                                 FROM       PROJECT
                                 WHERE      DNUM=5))
                AND
                NOT EXISTS      (SELECT     *
                                 FROM       WORKS_ON C
                                 WHERE      C.ESSN=SSN
                                            AND
                                            C.PNO=B.PNO));
```

8.5.5 Explicit Sets and Renaming of Attributes in SQL

We can put a explicit set of values in the **WHERE** clause.

(Q17:) Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

```
SELECT   DISTINCT ESSN
FROM     WORKS_ON
WHERE    PNO IN (1, 2, 3);
```

Table Name can be renamed (aliasing). Similarly, attribute name can be renamed too.

(Q8A:) Retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME.

```
SELECT   E.LNAME AS EMPLOYEE_NAME, S.LNAME AS
         SUPERVISOR_NAME
FROM     EMPLOYEE AS E, EMPLOYEE AS S
WHERE    E.SUPERSSN=S.SSN;
```

8.5.6 Joined Tables in SQL

In the **FROM** clause, we can have not only base tables (stored on the disk) but also **Joined tables**.

(Q1A:) Retrieve the name and address of every employee who works for the 'Research' department.

```
SELECT   FNAME, LNAME, ADDRESS
FROM     (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
WHERE    DNAME='Research';

SELECT   FNAME, LNAME, ADDRESS
FROM     (EMPLOYEE NATURAL JOIN
         (DEPARTMENT AS DEPT (DNAME, DNO, MSSN,
         MSDATE)))
```

WHERE DNAME='Research';

(Q8B:) Retrieve the last name of each employee and his or her supervisor (including NULL), while renaming the resulting attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME.

SELECT E.LNAME **AS** EMPLOYEE_NAME, S.LNAME **AS** SUPERVISOR_NAME
FROM (EMPLOYEE **AS** E **LEFT OUTER JOIN**[13] EMPLOYEE **AS** S **ON** E.SUPERSSN=S.SSN);

One of the tables in a join may itself be a joined table.

(Q2A): For each project located in 'Stafford', list the project number, the controlling department, and the department manager's last name, address, and birthdate.

SELECT PNUMBER, DNUM, LNAME, ADDRESS, BDATE
FROM ((PROJECT **JOIN** DEPARTMENT **ON** DNUM= DNUMBER) **JOIN** EMPLOYEE **ON** MGRSSN=SSN)
WHERE PLOCATION='Stafford';

8.5.7 Aggregate Functions in SQL

(Q19:) Find the sum of salary of all employees, the maximum salary, the minimum salary, and the average salary.

SELECT **SUM** (SALARY), **MAX** (SALARY), **MIN** (SALARY), **AVG** (SALARY)
FROM EMPLOYEE;

(Q20:) Find the sum of salary of all employees of the 'Research' department, as well as the the maximum salary, the minimum salary, and the average salary in this department.

[13]The key word OUTER may be omitted in LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN

```
SELECT   SUM (SALARY), MAX (SALARY), MIN (SALARY),
         AVG (SALARY)
FROM     EMPLOYEE, DEPARTMENT
WHERE    DNO=DNUMBER AND DNAME='Research';
```

Retrieve the total number of employees in the company **(Q21)** and the number of employees in the 'Research' department **(Q22)**.

```
SELECT   COUNT (*)
FROM     EMPLOYEE;

SELECT   COUNT (*)
FROM     EMPLOYEE, DEPARTMENT;
WHERE    DNO=DNUMBER AND DNAME='Research';
```

(Q23:) Count the number of distinct salary values in the database.

```
SELECT  COUNT (DISTINCT SALARY)
FROM    EMPLOYEE;
```

- If without **DISTINCT** in Q23, duplicate values will not be eliminated.
- If an employee's salary is **NULL**, it will **not** be counted. NULL values are discarded when evaluating an aggregate function.

(Q5:) Retrieve the names of all employees who have two or more dependents.

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    (SELECT  COUNT (*)
          FROM    DEPENDENT
          WHERE   SSN=ESSN) ≥ 2;
```

8.5.8 Groupings: The GROUP BY and HAVING Clauses

- Aggregate functions may be applied to each subgroups of tuples in a table (relation).
- Some attribute(s) are used to partition a relation to subgroups of tuples, called the **grouping attribute(s)**. Each tuple within a subgroup should have the same value(s) on the grouping attribute(s).

(Q24:) For each department, retrieve the department number, the number of employees in the department, and their average salary.
. Result is shown in Figure 8.6(a) (Fig 8.4(a) on e3)

```
SELECT    DNO, COUNT (*), AVG (SALARY)
FROM      EMPLOYEE
GROUP BY  DNO;
```

(Q25:) For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
SELECT    PNUMBER, PNAME, COUNT (*)
FROM      PROJECT, WORKS_ON
WHERE     PNUMBER=PNO
GROUP BY  PNUMBER, PNAME;
```

(Q26:) For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.
Result is shown in Fig 8.6(b) (Fig 8.4(b) on e3)

```
SELECT    PNUMBER, PNAME, COUNT (*)
FROM      PROJECT, WORKS_ON
WHERE     PNUMBER=PNO
GROUP BY  PNUMBER, PNAME
HAVING    COUNT (*) > 2;
```

(Q27:) For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
SELECT      PNUMBER, PNAME, COUNT (*)
FROM        PROJECT, WORKS_ON, EMPLOYEE
WHERE       PNUMBER=PNO AND ESSN=SSN AND DNO=5
GROUP BY    PNUMBER, PNAME;
```

The conditions in **WHERE** clause are evaluated first, to select individual tuples; the **HAVING** clause is applied later, to select individual groups of tuples.

(Q28:) For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

INCORRECT –

```
SELECT      DNUMBER, COUNT (*)
FROM        DEPARTMENT, EMPLOYEE
WHERE       DNUMBER=DNO AND SALARY > 40000
GROUP BY    DNUMBER
HAVING      COUNT (*) > 5;
```

CORRECT –

```
SELECT      DNUMBER, COUNT (*)
FROM        DEPARTMENT, EMPLOYEE
WHERE       DNUMBER=DNO AND SALARY > 40000 AND
            DNO IN (SELECT      DNO
                    FROM        EMPLOYEE
                    GROUP BY    DNO
                    HAVING      COUNT (*) > 5)
GROUP BY    DNUMBER
```

8.5.9 Discussion and Summary of SQL Queries

SELECT <attribute and function list>

FROM <table list>

[**WHERE** <condition>]

[**GROUP BY** <groupping attribute(s)>]

[**HAVING** <group condition>]

[**ORDER BY** <attribute list>];

8.6 Insert, Delete, and Update Statements in SQL

8.6.1 The INSERT Command

U1: **INSERT INTO** EMPLOYEE
VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest, Katy, TX', 'M', 37000, '987654321', 4);

U1A: **INSERT INTO** EMPLOYEE (FNAME, LNAME, DNO, SSN)
VALUES ('Richard', 'Marini', 4, '653298653'),
('Robert', 'Hatcher', 5, '623986996');

U2: **INSERT INTO** EMPLOYEE (FNAME, LNAME, SSN, DNO)
VALUES ('Robert', 'Hatcher', '980760540', 2);
(* U2 is rejected if referential integrity checking is provided by DBMS *)

U2A: **INSERT INTO** EMPLOYEE (FNAME, LNAME, DNO)
VALUES ('Robert', 'Hatcher', 5);
(* U2A is rejected if NOT NULL checking is provided by DBMS *)

U3A: **CREATE TABLE** DEPTS_INFO
(DEPT_NAME VARCHAR(15),
NO_OF_EMPS INTEGER,
TOTAL_SAL INTEGER);

```
U3B: INSERT INTO   DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,
                   TOTAL_SAL)
     SELECT        DNAME, COUNT (*), SUM (SALARY)
     FROM          (DEPARTMENT JOIN EMPLOYEE ON
                   DNUMBER=DNO)
     GROUP BY      DNAME;
```

8.6.2 The DELETE Command

```
U4A: DELETE FROM   EMPLOYEE
     WHERE         LNAME='Brown';

U4B: DELETE FROM   EMPLOYEE
     WHERE         SSN='123456789';

U4C: DELETE FROM   EMPLOYEE
     WHERE         DNO IN (SELECT  DNUMBER
                           FROM    DEPARTMENT
                           WHERE   DNAME='Research');

U4D: DELETE FROM   EMPLOYEE;
```

8.6.3 The Update Command

```
U5:  UPDATE   PROJECT
     SET      PLOCATION='Bellaire', DNUM=5
     WHERE    PNUMBER=10;

U6:  UPDATE   EMPLOYEE
     SET      SALARY=SALARY * 1.1
     WHERE    DNO IN  (SELECT  DNUMBER
                       FROM    DEPARTMENT
                       WHERE   DNAME='Research');
```

8.7 Specifying Genaral Constraints as Assertions

To specify any general constraints which could not be specified via declarative assertions discussed in Section 8.1.2

(A1:) To specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee work for".

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT *
                     FROM    EMPLOYEE E, EMPLOYEE M,
                             DEPARTMENT D
                     WHERE   E.SALARY>M.SALARY AND
                             E.DNO=D.DNUMBER AND
                             D.MGRSSN=M.SSN));
```

8.8 Views (Virtual Tables) in SQL

8.8.1 Concept of a View in SQL

- A view does not necessary exist in physical form, in contrast to base tables whose tuples are actually stored in the database.
- If frequently retrieve the employee name and the project names that the employee works on. We can define a view which is the result of **JOIN** of EMPLOYEE, WORKS_ON, and PROJECT tables. Then, we can have a single-table retrievals rather than as retrievals involving two joins on three tables.
- A view is always up to date.

8.8.2 Specification of Views in SQL

```
V1:  CREATE VIEW  WORKS_ON1
     AS  SELECT   FNAME, LNAME, PNAME, HOURS
         FROM     EMPLOYEE, PROJECT, WORKS_ON
         WHERE    SSN=ESSN AND PNO=PNUMBER;
```

WORKS_ON1

FNAME	LNAME	PNAME	HOURS

V2: **CREATE VIEW** DEPT_INFO (DEPT_NAME, NO_OF_EMPS, TOTAL_SAL)
AS **SELECT** DNAME, **COUNT** (*), **SUM** (SALARY)
FROM DEPARTMENT, EMPLOYEE
WHERE DNUMBER=DNO
GROUP BY DNAME;

DEPT_INFO

DEPT_NAME	NO_OF_EMPS	TOTAL_SAL

(QV1:) Retrieve the last name and first name of all employees who work on 'ProductX'.

SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNAME='ProductX';

If we do not need a view any more.

DROP VIEW WORKS_ON1;

8.8.3 View Implementation and View Update

Two main approaches to implement a view for querying are suggested.

- **Query Modification:** Modify the view query into a query on the underlying base tables (not efficient). For example, the query **QV1** would be modified by DBMS to

```
SELECT    FNAME, LNAME
FROM      EMPLOYEE, PROJECT, WORKS_ON
WHERE     SSN=ESSN AND PNO=PNUMBER
          AND PNAME='ProductX';
```

- **View Materialization:** Create a temporary view table when the view is first queried. (efficient but need some mechanisms to update the view table automatically when the defining base tables are updated).

A view update is feasible when only one possible update on the base tables can accomplish the desired update effect on the view.

- A view defined by a single base table without any aggregate functions is updatable if the view attributes contain the primary key (or candidate key) of the base table, because this maps each view tuple to a single base tuple.
- Views defined on multiple base tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

To update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY' on the view WORKS_ON1.

```
UV1:  UPDATE   WORKS_ON1
      SET      PNAME='ProductY'
      WHERE    LNAME='Smith' AND FNAME='John' AND
               PNAME='ProductX';
```

We have two possible updates on the base tables corresponding to UV1.

```
(a):  more likely
      UPDATE   WORKS_ON
      SET      PNO =     (SELECT PNUMBER FROM PROJECT
                         WHERE PNAME='ProductY')
      WHERE    ESSN IN   (SELECT SSN FROM EMPLOYEE
                         WHERE LNAME='Smith' AND
                         FNAME='John')
```

AND
PNO **IN** (**SELECT** PNUMBER **FROM** PROJECT
WHERE PNAME='ProductX');

(b): less likely (has side effect)
UPDATE PROJECT
SET PNAME='ProductY'
WHERE PNAME='ProductX';

To update the views defined using grouping and aggregate functions may not make much sense.

UV2: **UPDATE** DEPT_INFO
SET TOTAL_SAL = 100000
WHERE DNAME='Research';

Chapter 9, More SQL: Assertions, Views, and Programming Techniques

9.2 Embedded SQL

SQL statements can be embedded in a general purpose programming language, such as C, C++, COBOL,...

9.2.1 Retrieving Single Tuples with Embedded SQL

- **exec sql include sqlca;**
  exec sql begin declare section;
  char first_name[NAMESIZE];
  char last_name[NAMESIZE];
  char ssn[10];
  exec sql end declare section;

  strcpy(ssn, "987987987");
  exec sql select fname, lname
  into :first_name, :last_name
  from employee
  where ssn = **:ssn;**
  if (sqlca.sqlcode == 0)
  printf("%s, %s", first_name, last_name);
  else printf("No matching employee");

- The embedded SQL statements is distinguished from the programming language statements by prefixing it with a command, EXEC SQL, so that a **preprocessor** can separate them from the host language code, and the SQL statements are terminated by a matching END-EXEC, or ";".

- **Host variable/shared variable:** Within embedded SQL statements, we can refer to program variables (we call them shared variables), which are prefixed by a ":" sign.

This allows shared variables and database objects, such as attributes and relations, to have the same names.

- The shared variables used in embedded SQL statements should be declared somewhere else in the program. The declaration should preceded by **exec sql begin declare section;** and ended by **exec sql end declare section;**.

- **SQL communication area (sqlca):** After each SQL statement is executed, the DBMS provides feedback on whether the statement worked properly. This information is returned, via a collection of variables, to an area called sqlca (memory) that is shared by the host programming language and the SQL DBMS.

  - **"exec sql include sqlca;"** should be put somewhere in the program, the SQL compiler will insert the sqlca variables in place of the "exec sql include sqlca;" statement.

  - A variable in sqlca is called **sqlcode** which returns the status of each SQL statement execution. **0** means a successful execution; **100** means no more data/not found; $< \mathbf{0}$ means errors.

  - another variable in sqlca is **sqlstate**, which is a string of 5 characters. A value of "00000" means no error or exception; other values indicates various errors or exceptions.

9.2.2 Retrieving Multiple Tuples with Embedded SQL Using Cursors

- **exec sql begin declare section;**
  char first_name[NAMESIZE];
  char last_name[NAMESIZE];
  exec sql end declare section;

  exec sql declare emp_dept **cursor for**
  select fname, lname

```
from    employee
where   dno=:dnumber;

exec sql open emp_dept;
while   (sqlca.sqlcode == 0){
        exec sql fetch emp_dept into :first_name, :last_name;
        if (sqlca.sqlcode == 0)
          printf("First Name: %s, Last Name: %s", first_name, last_name);
        else
          printf("ERROR MESSAGE");
}
exec sql close emp_dept;
```

- The **Cursor** structure represents an area in memory allocated for temporarily storing and processing the results of an SQL SELECT statement.
  - The cursor-name (emp_dept in above example) is the name assigned to the cursor structure.
  - The **select** statement defines the query.
  - The **declare cursor** statement is declarative; the query is not executed at this time.
  - The **open** statement open the cursor, and the **select** statement defined in declare cursor statement is executed and the set of tuples is stored in the cursor structure. This open statement will set a **pointer (current pointer)** pointing to the position before the first row of the query result.
  - The **fetch** statement fetches one row from the result into the host variables and moves the pointer to the next row in the result of the query.
  - The **close** statement closes the cursor.
- **Update command in embedded SQL:**

- Update without cursor structure:

```
exec sql update   employee
set               salary = salary * 1.1
where             dno=5;
```

- Update with cursor structure:

```
exec sql declare emp_d5 cursor for
select   ssn, salary
from     employee
where    dno=5
for update of salary;

exec sql update   employee
set               salary = salary * 1.1
where             current of emp_d5;
```

  * The cursor structure must be opened and positioned (using FETCH) on a row before the UPDATE command can be executed.
  * Each execution of the UPDATE statement updates one row - the row at which the cursor is positioned.
  * The only columns that can be updated are those listed in the FOR UPDATE OF clause of the DECLARE CURSOR statement.
  * The cursor is not moved by the execution of the UPDATE statement. The FETCH statement moves the cursor.

9.2.3 Specifying Queries at Runtime Using Dynamic SQL

- Dynamic SQL allows a program to form an SQL statement during execution.
- Parparing a statement:
  - **exec sql prepare** update_salary **from**
    " update employee

set salary = salary * (1 + ?/100)

where dno = ?";

- The question mark indicates that when the statement is executed, the value of the shared variable will be used.

- Executing prepared SQL:
  - Statements other than SELECT statements, and SELECT statements that return only a single row, are executed with the EXECUTE statement.
  - **exec sql execute** update_salary **using** :rate, :dnumber;

- Using prepare statement and declare cursor statement:
  - **exec sql prepare** update_salary **from**

    " update employee

    set salary = salary * (1 + ?/100)

    where **current of** emp";

    exec sql declare cursor emp **for**

    select ssn, salary

    where dno = :dnumber

    for update of salary;

    exec sql open emp;

    exec sql fetch emp into :ssn, :salary;

    exec sql execute update_salary **using** :rate;

Chapter 10, Functional Dependencies and Normalization for Relational Databases

- We need some formal measure of why the choice of attributes for a relation schema may be better than another.
- Functional dependencies among attributes within a relation is the main tool for formally measuring the appropriateness of attribute groupings into relation schemas.

10.1 Informal Design Guidelines for Relation Schemas

Four informal measures of quality for relation schema design.

- Semantics of the attributes.
- Reducing the redundant values in tuples.
- Reducing the null values in tuples.
- Disallowing the possibility of generating spurious tuples.

10.1.1 Semantics of the Relation Attributes

- The easier it is to explain the semantics of the relation, the better the relation schema design will be.
- GUIDELINE 1: Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, the meaning tends to be clear. Otherwise, the relation corresponds to a mixture of multiple entities and relationships and hence becomes semantically unclear.
- Example: A relation involves two entities – poor design.
  EMP_DEPT

ENAME	SSN	BDATE	ADDREESS	DNUMBER	DNAME	DMGRSSN

10.1.2 Redundant Information in Tuples and Update Anomalies

- Grouping attributes into relation schemas has a significant effect on storage space. Compare two base relations EMPLOYEE and DEPARTMENT in Figure 10.2 (Fig 14.2 on e3) to an EMP_DEPT base relation in Figure 10.4 (Fig 14.4 on e3).

- Update anomalies for base relations EMP_DEPT and EMP_PROJ in Figure 10.4 (Fig 14.4 on e3).

  - Insertion anomalies: For EMP_DEPT relation in Figure 10.4 (Fig 14.4 on e3).

    * To insert a new employee tuple, we need to make sure that the values of attributes DNUMBER, DNAME, and DMGRSSN are consistent to other employees (tuples) in EMP_DEPT.

    * It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation.

  - Deletion anomalies: If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

  - Modification anomalies: If we update the value of MGRSSN in a particular department, we must to update the tuples of all employees who work in that department; otherwise, the database will become inconsistent.

- GUIDELINE 2: Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure the programs that update the database will operate correctly.

- It is advisable to use anomaly-free base relations and to specify views that include the JOINs for placing together the attributes frequently referenced to improve the performance.

10.1.3 Null Values in Tuples

- Having null values in tuples of a relation not only wastes storage space but also makes the interpretation more difficult.
- GUIDELINE 3: As far as possible, avoid placing attributes in a base relation whose values may frequently be null. If nulls are unavoidable, make sure that they apply in exceptional cases only and do not apply to majority of tuples in the relation.

10.1.4 Generation of Spurious Tuples

- GUIDELINE 4: Design relation schemas so that they can be JOINed with equality conditions on attributes that are either primary keys or foreign keys in a way that guarantees that no spurious tuples are generated. Do not have the relations that contains matching attributes other than foreign key - primary key combinations. If such relations are unavoidable, do not join them on such attributes.
- For example, decomposing EMP_PROJ in Figure 10.4 (Fig 14.4 on e3) to EMP_LOCS and EMP_PROJ1 in Figure 10.5 (Fig 14.5 on e3) is undesirable because spurious tuples will be generated if NATURAL JOIN operation is performed (see Figure 10.6 (Fig 14.6 on e3)).

10.2 Functional Dependencies

Functional dependencies are the main tool for defining normal forms of relation schemas.

10.2.1 Definition of Functional Dependency

- A **functional dependency (abbreviated as FD or f.d.)**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of $R = \{A_1, A_2, \ldots, A_n\}$ specifies a *constraint* on the possible tuples that can form a relation state r of R.
  The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, we must also have $t_1[Y] = t_2[Y]$.
- $X \rightarrow Y$: *X functionally determines Y* or *Y is functionally dependent on X*

- X functionally determines Y in a relation schema R if and only if , whenever two tuples of $r(R)$ agree on their X-values, they must necessarily agree on their Y-values.
  - If X is a **candidate key** of R, this implies that $X \rightarrow Y$ for any subset of attributes Y of R.
  - If $X \rightarrow Y$ in R, this does not say whether or not $Y \rightarrow X$ in R.
- A functional dependency is a constraint that any relation extensions $r(R)$ must satisfy the functional dependency constraint at all times.
- Figure 10.3 (Fig 14.3 on e3) shows the **diagrammatic notation** for FDs.

10.2.2 Inference Rules for Functional Dependencies

- We denote by F the set of functional dependencies that are specified on relation schema R. Typically, the schema designer specifies the FDs that are semantically obvious.
- It is practically impossible to specify all possible FDs that may hold in a relation schema. The set of all such FDs is called the **closure** of F and is denoted by F^+.
- For example, let $F = \{SSN \rightarrow \{ENAME, BDATE, ADDRESS, DNUMBER\}, DNUMBER \rightarrow \{DNAME, DMGRSSN\}\}$. The following additional FDs can be *inferred* from F.
  $SSN \rightarrow \{DNAME, DMGRSSN\}$,
  $SSN \rightarrow SSN$,
  $DNUMBER \rightarrow DNAME$
- $\forall$ FD $X \rightarrow Y \in F^+$, $X \rightarrow Y$ should hold in every relation state r that is a legal extension of R.
- 6 well-known **Inference rules** that can be used to infer new dependencies from a given set of dependencies F. ($F \models X \rightarrow Y$ denotes the FD $X \rightarrow Y$ is inferred from F.)
  - **IR1 (reflexive rule):** If $X \supseteq Y$, then $X \rightarrow Y$.
  - **IR2 (augmentation rule):** $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

- **IR3 (transitive rule):** $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
- **IR4 (decomposition, or projective rule):** $\{X \rightarrow YZ\} \models X \rightarrow Y$.
- **IR5 (union, or additive rule):** $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.
- **IR6 (pseudotransitive rule):** $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

- A functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**.

- **Armstrong's inference rules:** IR1, IR2, and IR3 are complete. That is, the set of dependencies F^+ can be determined from F by using only inference rules IR1 through IR3.

- The proofs for inference rules:
  - **Proof of IR1:** If $X \supseteq Y$, then $X \rightarrow Y$.
    Suppose that $X \supseteq Y$ and that two tuples t_1 and t_2 exist in some relation instance r of R such that $t_1[X] = t_2[X]$. Then $t1_{[}Y] = t_2[Y]$ because $X \supseteq Y$; hence, $X \rightarrow Y$ must hold in r.
  - **Proof of IR2:** $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.
    Assume that $X \rightarrow Y$ holds in a relation instance r of R but that $XZ \rightarrow YZ$ does not hold. Then there must exist two tuples t_1 and t_2 in r such that (1) $t_1[X] = t_2[X]$, (2) $t_1[Y] = t_2[Y]$, (3) $t_1[XZ] = t_2[XZ]$, and (4) $t_1[YZ] \neq t_2[YZ]$. This is not possible because from (1) and (3) we deduce (5) $t_1[Z] = t_2[Z]$, and from (2) and (5) we deduce (6) $t_1[YZ] = t_2[YZ]$, contradicting (4).
  - **Proof of IR3:** $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
    Assume that (1) $X \rightarrow Y$ and (2) $Y \rightarrow Z$ both hold in a relation r. Then for any two tuples t_1 and t_2 in r such that $t_1[X] = t_2[X]$, we must have (3) $t_1[Y] = t_2[Y]$, from assumption (1); hence we must also have (4) $t_1[Z] = t_2[Z]$, from (3) and assumption (2); hence $X \rightarrow Z$ must hold in r.

- All other inference rules (IR4 - IR6) can be proved by using IR1 through IR3.
  - **Proof of IR4:** $\{X \rightarrow YZ\} \models X \rightarrow Y$.
    1. $X \rightarrow YZ$ (given)

2. $YZ \rightarrow Y$ (by IR1)

3. $X \rightarrow Y$ (by IR3 and 1 and 2).

– **Proof of IR5:** $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.

1. $X \rightarrow Y$ (given).
2. $X \rightarrow Z$ (given).
3. $XZ \rightarrow YZ$ (by IR2 and 1)
4. $X \rightarrow XZ$ (by IR2 and 2, note that XX = X)
5. $X \rightarrow YZ$ (by IR3 and 3 and 4)

– **Proof of IR6:** $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

1. $X \rightarrow Y$ (given)
2. $WY \rightarrow Z$ (given)
3. $WX \rightarrow WY$ (by IR2 and 1)
4. $WX \rightarrow Z$ (by IR3 and 2 and 3)

- A systematic way to determine F^+: (1) determine each set of attributes X that appears as a left-hand side of some FDs in F, (2) determine the set of all attributes that are dependent on X.

- **closure of X under F (denote X_F^+):** The set of all attributes are functionally determined by X under F.

  – **Algorithm 10.1** Determining X_F^+, the closure of X under F

  $X_F^+ := X$;

  repeat

  $oldX_F^+ := X_F^+$;

  for each functional dependency $Y \rightarrow Z$ in F do

  if $X_F^+ \supseteq Y$ then $X_F^+ := X_F^+ \cup Z$;

  until $(X_F^+ = oldX_F^+)$;

  – Example: Let $R(A, B, C, D, E, F)$ and

  $F = \{A \rightarrow D, B \rightarrow EF, AB \rightarrow C\}$.

  Then

What is the closure of $\{A\}$ under F ?

$X_F^+ = A_F^+ = A$; $oldX_F^+ = A$

$X_F^+ = AD \qquad oldX_F^+ = AD$

$X_F^+ = AD$

What is the closure of $\{B\}$ under F ?

$X_F^+ = B_F^+ = B$; $oldX_F^+ = B$

$X_F^+ = BEF \qquad oldX_F^+ = BEF$

$X_F^+ = BEF$

What is the closure of $\{A, B\}$ under F ?

$X_F^+ = (AB)_F^+ = AB$; $oldX_F^+ = AB$

$X_F^+ = ABDEFC \qquad oldX_F^+ = ABDEFC$

$X_F^+ = ABDEFC$

What is F^+ ?

$\{A\}_F^+ = \{A, D\}$	3 functional dependencies
$\{B\}_F^+ = \{B, E, F\}$	7 functional dependencies
$\{A, B\}_F^+ = \{A, B, D, E, F, C\}$	31 functional dependencies
$F^+ = \{A\}_F^+ \cup \{B\}_F^+ \cup \{A, B\}_F^+$	41 functional dependencies

– Another example, let $R(A, B, C, D, E, I)$ and

$F = \{A \to D,\ AD \to E,\ BI \to E,\ CD \to I,\ E \to C\}$.

Then

What is the closure of $\{A, E\}$ under F ?

$X_F^+ = (AE)_F^+ = AE$; $oldX_F^+ = AE$

$X_F^+ = AEDC \qquad oldX_F^+ = AEDC$

$X_F^+ = AEDCI \qquad oldX_F^+ = AEDCI$

$X_F^+ = AEDCI$

- Testing membership with respect to F, i.e., does $F \models X \rightarrow Y$?
  - **Algorithm 10.1A** MEMBER($X \rightarrow Y,\ F$) : BOOLEAN

    Input: A set F of functional dependencies and a functional dependency $X \rightarrow Y$.

    Output: TRUE if $F \models X \rightarrow Y$; FALSE otherwise.

    begin

    If $X_F^+ \supseteq Y$ then return TRUE else return FALSE

    End
  - Example: let $F = \{A \rightarrow D,\ AD \rightarrow E,\ BI \rightarrow E,\ CD \rightarrow I,\ E \rightarrow C\}$.

    Then does $F \models \{A, B\} \rightarrow \{E\}$?

    Find $\{A, B\}_F^+$: $(AB)_F^+ = ABDECI$

    Since $\{A, B, D, E, C, I\} \supseteq \{E\} \Rightarrow F \models \{A, B\} \rightarrow \{E\}$

10.2.3 Equivalence of Sets of Functional Dependencies

- A set of functional dependencies E is **covered by** a set of functional dependencies F if every FD in E is also in F^+.
- Two sets of functional dependencies E and F are **equivalent** if $E^+ = F^+$.
- Alternatively, E is equivalent to F if both E covers F and F covers E hold.
- Algorithm for determining whether F covers E:
  - **Algorithm 10.1B** Determining whether F covers E.

    for each functional dependency $X \rightarrow Y \in E$

    do if (not MEMBER($X \rightarrow Y,\ F$))

    then return false;

    return true;
- For example, let

  $F_1 = \{A \rightarrow BC,\ A \rightarrow D,\ CD \rightarrow E\}$

$F_2 = \{A \rightarrow BCE,\ A \rightarrow ABD,\ CD \rightarrow E\}$

$F_3 = \{A \rightarrow BCDE\}$

Then

Is F_1 equivalent to F_2 ?

Since $F_1 \models F_2$ and $F_2 \models F_1 \Rightarrow F_1 \equiv F_2$

Is F_1 equivalent to F_3 ?

$F_1 \models F_3$ but $F_3 \not\models F_1 \Rightarrow F_1 \not\equiv F_3$

10.2.4 Minimal Sets of Functional Dependencies

- A set of functional dependencies F is **minimal** if it satisfies the following conditions:
  - 1. Single attribute on right-hand side for every FD in F.
  - 2. $F - \{X \rightarrow A\} \cup \{Y \rightarrow A\}$ is not equivalent to F, for all Y that is a proper subset of X and for all $X \rightarrow A$ in F.
  - 3. $F - \{X \rightarrow A\}$ is not equivalent to F, for all $X \rightarrow A$ in F.
- A **minimal cover** of a set of functional dependencies F is a minimal set of dependencies F_{min} that is equivalent to F.
- Algorithm for finding a minimal cover G for F.
  - **Algorithm 10.2** Finding a minimal cover G for F
    1. Set $G := F$.
    2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ in G by the n functional dependencies $X \rightarrow A_1,\ X \rightarrow A_2,\ \ldots,\ X \rightarrow A_n$.
    3. For each functional dependency $X \rightarrow A$ in G

       for each attribute B that is an element of X

       if $((G - \{X \rightarrow A\}) \cup \{(X - \{B\}) \rightarrow A\})$ is equivalent to G,

       then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in G.
    4. For each remaining functional dependency $X \rightarrow A$ in G

if $(G - \{X \to A\})$ is equivalent to G,

then remove $X \to A$ from G.

- Example: find a minimal cover G of $F = \{A \to BC,\ B \to C,\ AB \to D\}$.
  - **Step 1:**

    Let $G = \{A \to BC,\ B \to C,\ AB \to D\}$.
  - **Step 2:**

    $G = \{A \to B,\ A \to C,\ B \to C,\ AB \to D\}$.
  - **Step 3:**
    - Can A be eliminated from $AB \to D$ from G?

      Let $G_1 = \{A \to B,\ A \to C,\ B \to C,\ B \to D\}$.

      Is $G_1 \equiv G$? $\Rightarrow$ *Is* G_1 *cover* G *and Is* G *cover* G_1? $\Rightarrow G_1 \models AB \to D$? *and* $G \models B \to D$?

      Since $(AB)^+_{G_1} = \{A, B, C, D\} \supseteq \{D\} \Rightarrow G_1 \models AB \to D$

      Since $(B)^+_G = \{B, C\} \not\supseteq \{D\} \Rightarrow G \not\models B \to D$

      Therefore, $G_1 \not\equiv G$ and A can not be eliminated from $AB \to D$ in G.
    - Can B be eliminated from $AB \to D$ from G?

      Let $G_1 = \{A \to B,\ A \to C,\ B \to C,\ A \to D\}$.

      Is $G_1 \equiv G$? $\Rightarrow$ *Is* G_1 *cover* G *and Is* G *cover* G_1? $\Rightarrow G_1 \models AB \to D$? *and* $G \models A \to D$?

      Since $(AB)^+_{G_1} = \{A, B, C, D\} \supseteq \{D\} \Rightarrow G_1 \models AB \to D$

      Since $(A)^+_G = \{A, B, C, D\} \supseteq \{D\} \Rightarrow G \models B \to D$

      Therefore, $G_1 \equiv G$ and B can be eliminated from $AB \to D$ in G.
  - **Step 4:** $G = \{A \to B,\ A \to C,\ B \to C,\ A \to D\}$.
    - Can $A \to B$ be eliminated from G?

      Let $G_1 = \{A \to C,\ B \to C,\ A \to D\}$.

      Is $G_1 \equiv G$? $\Rightarrow$ *Is* G_1 *cover* G *and Is* G *cover* G_1? $\Rightarrow G_1 \models A \to B$?

      Since $(A)^+_{G_1} = \{A, C, D\} \not\supseteq \{B\} \Rightarrow G_1 \not\models A \to B$

      Therefore, $A \to B$ can not be eliminated from G.

    - Can $A \rightarrow C$ be eliminated from G?
    - Can $B \rightarrow C$ be eliminated from G?
    - Can $A \rightarrow D$ be eliminated from G?

- Another Example: Find a minimal cover G for $F = \{A \rightarrow C,\ AC \rightarrow J,\ AB \rightarrow DE,\ AB \rightarrow CDI,\ C \rightarrow M,\ A \rightarrow M\}$.

10.2.5 Candidate Keys and Superkeys

Let $R = (U, F)$ be a relation schema, where U is the set of attributes and F is the set of functional dependencies.

- A subset X of U is a **superkey** for a relation schema R if $(X \rightarrow U)$ is in F^+, i.e., $X_F^+ = U$.

- A subset X of U is a **candidate key** for R if X is a superkey and no proper subset Y of X is a superkey.
  That is, if X is a **candidate key** , then $(X_F^+ = U)$ and $(\nexists Y \textit{ such that } Y \subset X \textit{ and } Y_F^+ = U)$.

- Example: Is (ABE) a superkey for $R = \{U, F\}$, where $U = \{A, B, C, D, E, I\}$ and $F = \{A \rightarrow D,\ AD \rightarrow E,\ BI \rightarrow E,\ CD \rightarrow I,\ E \rightarrow C\}$
  - Is $F \models X \rightarrow U$?
    $(ABE)_F^+ = ABEDCI = U$, therefore, $ABE \rightarrow U$ and ABE is a superkey for R.

- Example: Is (ABE) a candidate key for R ? Where R is as previous example.
  - We know ABE is a superkey.
  - $A_F^+ = ADECI \neq U$
    $B_F^+ = B \neq U$
    $E_F^+ = EC \neq U$
    $AB_F^+ = ABDECI = U \Rightarrow ABE$ is not a candidate key.
    $AE_F^+ =$

$BE_F^+ =$

$ABE_F^+ =$

- How about finding all candidate keys for $R = \{U, F\}$, where $U = \{A, B, C, D, E\}$ and $F = \{AB \rightarrow E,\ E \rightarrow AB,\ EC \rightarrow D\}$.
  - Find all candidate keys containing 1 attribute.
  - Find all candidate keys containing 2 attributes.
  - Find all candidate keys containing 3 attributes.
  - Find all candidate keys containing 4 attributes.
  - Find all candidate keys containing 5 attributes.

10.3 Normal Forms Based on Primary Keys

10.3.1 Normalization of Relations

- The **normalization process** takes a relation schema through a series of tests to "certify" whether it satisfies a certain **normal form**. If the schema does not meet the normal form test, the relation is decomposed into smaller relation schemas that meet the tests.
- The purpose of **normalization** is to analyze the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies.
- 1NF, 2NF, 3NF, and BCNF are based on the functional dependencies among the attributes of a relation.
- 4NF is based on multivalued dependencies; 5NF is based on join dependencies.
- Database design as praticed in industry today pays particular attention to normalization only up to 3NF or BCNF (sometimes 4NF).

- An attribute of relation schema R is called a **prime attribute** of R if it is a member of some candidate key of R. An attribute is called **nonprime** if it is not a prime attribute.

10.3.4 First Normal Form

- 1NF: the domain of an attribute must include only *atomic values* and the value of any attribute in a tuple must be a *single value* from the domain of that attribute.
- Example: Suppose we extend the DEPARTMENT relation in Figure 10.1 (Fig 14.1 on e3) by including the DLOCATIONS attributes as in Figure 10.8 (Fig 14.8 on e3).
  - Two cases:
    * Case1: the domain of DLOCATIONS contains atomic values, but some tuples can have a set of these values. In this case, DLOCATIONS is not functionally dependent on DNUMBER.
    * Case2: the domain of DLOCATIONS contains sets of values and hence is nonatomic. In this case, $DNUMBER \rightarrow DLOCATIONS$.
    * Both cases violate the first normal form.
  - There are three decomposition (normalization) approaches for first normal form:
    * 1. Decompose DEPARTMENT(DNAME, <u>DNUMBER</u>, DMGRSSN, DLOCATIONS) into two relations DEPARTMENT(DNAME, <u>DNUMBER</u>, DMGRSSN) and DEPT_LOCS(<u>DNUMBER, DLOCATIONS</u>)
    * 2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a department, as shown in Figure 10.8(c) (Fig 14.8(c) on e3).
      Introduce *redundancy*.
    * 3. If it is known that at most three locations can exist for a department, then replace DLOCATIONS to three attributes: DLOCATION1, DLOCATION2, and DLOCATION3.
      Introduce *null* values.

10.3.5 Second Normal Form

- 2NF is based on the concept of *fully functional dependency*.
- $X \rightarrow Y$ is a **full functional dependency** if for any attribute $A \in X$, $(X - \{A\}) \not\rightarrow Y$.
- $X \rightarrow Y$ is a **partial functional dependency** if for some attribute $A \in X$, $(X - \{A\}) \rightarrow Y$.
- 2NF: a relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.
- For example, the EMP_PROJ relation in Figure 10.10(a) (Fig 14.10(a) on e3) is in 1NF but not in 2NF.
  - Decompsition (Normalization) to 2NF:
    The FD1, FD2, and FD3 in Figure 10.10(a) (Fig 14.10(a) on e3) lead to the decomposition of EMP_PROJ into three relation schemas EP1, EP2, EP3.

10.3.6 Third Normal Form

- 3NF is based on the concept of *transitive dependency*.
- $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.
- 3NF: a relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key.
- For example, the EMP_DEPT relation in Figure 10.10(b) (Fig 14.10(b) on e3) is in 2NF but not in 3NF.
  - Decomposition (Normalization) to 3NF:
    Decompose EMP_DEPT into ED1 and ED2 as shown in Figure 10.10(b) (Fig 14.10(b) on e3).

10.4 General Definitions of Second and Third Normal Forms

- In previous section, the definitions of 2NF and 3NF do not take other candidate keys into consideration. In this section, we give more general definitions of 2NF and 3NF that take all candidate keys of a relation into account.

10.4.1 General Definition of Second Normal Form

- 2NF: a relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on every key of R.

- For example, the relation schema LOTS shown in Figure 10.11(a) (Fig 14.11(a) on e3), which has two candidate keys: PROPERTY_ID# and {COUNTY_NAME, LOTS#}. FD3 in LOTS violates 2NF. LOTS is decomposed into two relation schemas LOTS1 and LOTS2 as shown in Figure 10.11(b) (Fig 14.11(b) on e3).

10.4.2 General Definition of Third Normal Form

- 3NF: a relation achema R is in 3NF if for any *nontrivial* functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

- For example, the FD4 of LOTS1 in Figure 10.11(b) (Fig 14.11(b) on e3) violates the 3NF because AREA is not a superkey and PRICE is not a prime attribute in LOTS1. LOTS1 is decomposed into two relation schemas LOTS1A and LOTS1B as shown in Figure 10.11(c) (Fig 14.11(c) on e3).

10.5 Boyce-Codd Normal Form

- BCNF is a stronger normal form than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF.

- BCNF: a relation schema R is in BCNF if whenever a nontrivial functional dependency $X \rightarrow A$ holds in R, then X is a superkey of R.

- For example, consider the LOTS relation schema in Figure 10.11 (Fig 14.11 on e3). If there is another functional dependency FD5: AREA → COUNTY_NAME.
  LOTS1A is still in 3NF but not in BCNF. LOTS1A can be decomposed into two relation schemas LOTS1AX and LOTS1AY as shown in Figure 10.12 (Fig 14.12 on e3). Notice that, FD2 is lost in this decomposition.

- In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in R with X not being a superkey and A being a prime attribute will R be in 3NF but not in BCNF. Figure 10.12(b) (Fig 14.12(b) on e3) shows the relation in the case.

- A relation not in BCNF should be decomposed so that to meet this property, while possibly forgoing the preservation of all functioal dependencies and also a test is necessary to determine whether the decomposition is nonadditive (lossless), i.e., it will not generate spurious tuples after a join. See the example in Figure 10.13 (Fig 14.13 on e3).

Chapter 11, Relational Database Design Algorithms and Further Dependencies

- Normal forms are *insufficient on their own* as a criteria for a good relational database schema design.
- The relations in a database must collectively satisfy two other properties - **dependency preservation property and lossless (or nonadditive) join property** - to qualify as a good design.

11.1 Properties of Relational Decompositions

11.1.1 Relation Decomposition and Insufficiency of Normal Forms

- The relational database design algorithms discussed in this chapter start from a single **universal relation schema** $R = \{A_1, A_2, \ldots, A_n\}$ that includes all the attributes of the database. Using the functional dependencies F, the algorithms decompose R into a set of relation schemas $DECOMP = \{R_1, R_2, \ldots, R_m\}$ that will become the relational database schema.
- Examine an individual relation R_i to test whether it is in a higher normal form does not guarantee a good design (decomposition); rather, *a set of relations* that together form the relation database schema must possess certain additional properties to ensure a good design.
  - **Attribute preservation** property: Each attribute in R will appear in at least one relation R_i in the decomposition so that no attributes are 'lost'; formally we have
    $$\bigcup_{i=1}^{m} R_i = R$$
  - **Dependency preservation** property: See Figure 10.12 (Fig 14.12 on e3). Discuss on 11.1.2
  - **Lossless (nonadditive) join** property: If we decompose EMP_PROJ in Figure 10.2 (Fig 14.2 on e3) to EMP_LOCS and EMP_PROJ1 in Figure 10.5 (Fig 14.5 on e3), it will violate the nonadditive join property. Discuss on 11.1.3

11.1.2 Decomposition and Dependency Preservation

- It is not necessary that the exact dependencies specified in F on R appear themselves in individual relations of the composition $DECOMP$. It is sufficient that the **union** of the dependencies that hold on individual relations in $DECOMP$ be **equivalent** to F.

- The **projection** of F on R_i, denoted by $\pi_{R_i}(F)$, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are contained in R_i.

- A decomposition $DECOMP = \{R_1, R_2, \ldots, R_m\}$ of R is **dependency-preserving** with respect to F if

$$((\pi_{R_1}(F)) \cup \ldots \cup (\pi_{R_m}(F)))^+ = F^+$$

- **claim 1:** It is always possible to find a dependency-preserving decomposition $DECOMP$ with respect to F such that each relation R_i in $DECOMP$ is in 3NF.

11.1.3 Decomposition and Lossless (Nonadditive) Joins

- A decomposition $DECOMP = \{R_1, R_2, \ldots, R_m\}$ of R has the lossless join property with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F, the following holds, where $*$ is the NATUAL JOIN of all the relations in $DECOMP$:

$$*(\pi_{R_1}(r), \ldots, \pi_{R_m}(r)) = r$$

- The decomposition of EMP_PROJ(SSN, PNUMBER, HOURS, ENAME, PNAME, PLOCATION) from Figure 10.3 (Fig 14.3 on e3) into EMP_LOCS(ENAME, PLOCATION) and EMP_ PROJ1(SSN, PNUMBER, HOURS, PNAME, PLOCATION) in Figure 10.5 (Fig 14.5 on e3) does not have the lossless join property.

- Another example: decompose R to R_1 and R_2 as follows.

R

A	B	C	D
1	2	3	4
1	2	3	5
4	2	3	4
3	1	4	2

R_1

A	B	C
1	2	3
4	2	3
3	1	4

R_2

B	C	D
2	3	4
2	3	5
1	4	2

Is R equal to $R_1 * R_2$?

$R_1 * R_2$			
A	B	C	D
1	2	3	4
1	2	3	5
4	2	3	4
4	2	3	5
3	1	4	2

- **Algorithm 11.1** Testing for the lossless (nonadditive) join property

  Input: A universal relation R, a decomposition $DECOMP = \{R_1, R_2, \ldots, R_m\}$ of R, and a set F of functional dependencies.

  - **1.** Create an initial matrix S with one row i for each relation R_i in $DECOMP$, and one column j for each attribute A_j in R.
  - **2.** Set $S(i, j) := b_{ij}$ for all matrix entries.
  - **3.** For each row i

    For each column j

    If R_i includes attribute A_j

    Then set $S(i, j) := a_j$
  - **4.** Repeat the following loop until a complete loop execution results in no changes to S

    For each $X \rightarrow Y$ in F

    For all rows in S which has the same symbols in the columns corresponding to attributes in X

    make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: if any of the rows has an "a" symbol for the column, set the other rows to the same "a" symbol in the column. If no "a" symbol exists for the attribute in any of the rows, choose one of the "b" symbols that appear in one of the rows for the attribute and set the other rows to that same "b" symbol in the column

- **5.** If a row is made up entirely of "a" symbols, then the decomposition has the lossless join property; otherwise it does not.

- Example:

  $R = \{SSN,\ ENAME,\ PNUMBER,\ PNAME,\ PLOCATION,\ HOURS\}$

  $F = \{SSN \rightarrow ENAME,\ PNUMBER \rightarrow \{PNAME,\ PLOCATION\},\ \{SSN,\ PNUMBER\} \rightarrow HOURS\}$

  $DECOMP = \{R_1, R_2, R_3\}$

  $R_1 = \{SSN,\ ENAME\}$

  $R_2 = \{PNUMBER,\ PNAME,\ PLOCATION\}$

  $R_3 = \{SSN,\ PNUMBER,\ HOURS\}$

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32} a_2	a_3	b_{34} a_4	b_{35} a_5	a_6

  - Try $SSN \rightarrow ENAME$:

    $a_1 \not\rightarrow b_{32}\ a_2$

  - Try $PNUMBER \rightarrow \{PNAME,\ PLOCATION\}$:

    $a_3 \not\rightarrow b_{34}\ a_4$

    $a_3 \not\rightarrow b_{35}\ a_5$

    In row 3, all symbols are a_i. The decomposition $DECOMP$ has the lossless join property.

- Another example:

  $R = \{A, B, C, D, E\}$

  $F = \{A \rightarrow C,\ B \rightarrow C,\ C \rightarrow D,\ DE \rightarrow C,\ CE \rightarrow A\}$

  $DECOMP = \{R_1, R_2, R_3, R_4, R_5\}$

$R_1 = \{A, D\}$

$R_2 = \{A, B\}$

$R_3 = \{B, E\}$

$R_4 = \{C, D, E\}$

$R_5 = \{A, E\}$

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{23}	b_{24}	b_{25}
R_3	b_{31}	a_2	b_{33}	b_{34}	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{53}	b_{54}	a_5

- For each FDs in F (first loop):
  - Try $A \rightarrow C$:

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{23} b_{13}	b_{24}	b_{25}
R_3	b_{31}	a_2	b_{33}	b_{34}	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{53} b_{13}	b_{54}	a_5

  - Try $B \rightarrow C$:

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{13}	b_{24}	b_{25}
R_3	b_{31}	a_2	b_{33} b_{13}	b_{34}	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{13}	b_{54}	a_5

* Try $C \rightarrow D$:

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{13}	$b\!\!\not\!\!/_{24}$ a_4	b_{25}
R_3	b_{31}	a_2	b_{13}	$b\!\!\not\!\!/_{34}$ a_4	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{13}	$b\!\!\not\!\!/_{54}$ a_4	a_5

* Try $DE \rightarrow C$:

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{13}	a_4	b_{25}
R_3	b_{31}	a_2	$b\!\!\not\!\!/_{13}$ a_3	a_4	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	$b\!\!\not\!\!/_{13}$ a_3	a_4	a_5

* Try $CE \rightarrow A$:

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{13}	a_4	b_{25}
R_3	$b\!\!\not\!\!/_{31}$ a_1	a_2	a_3	a_4	a_5
R_4	$b\!\!\not\!\!/_{41}$ a_1	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	a_3	a_4	a_5

– The third row is made up entirely of a_i symbols. The decomposition $DECOMP$ has the lossless join property.

- Another example:

  $R = \{A, B, C\}$

  $F = \{AB \rightarrow C,\ C \rightarrow B\}$

  $DECOMP = \{R_1, R_2\}$

  – If $R_1 = \{A, B\}$ and $R_2 = \{B, C\}$

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	b_{21}	a_2	a_3

* For each FDs in F (first loop):
  - Try $AB \rightarrow C$:

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	b_{21}	a_2	a_3

  - Try $C \rightarrow B$:

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	b_{21}	a_2	a_3

  - Complete loop execution results in no changes to the matrix and the matrix does not contain a row with all a_i symbols. This decomposition does not have lossless join property.

– If $R_1 = \{A, C\}$ and $R_2 = \{B, C\}$

	A	B	C
R_1	a_1	b_{12}	a_3
R_2	b_{21}	a_2	a_3

* For each FDs in F (first loop):
  - Try $AB \rightarrow C$:

	A	B	C
R_1	a_1	b_{12}	a_3
R_2	b_{21}	a_2	a_3

  - Try $C \rightarrow B$:

	A	B	C
R_1	a_1	b_{12} a_2	a_3
R_2	b_{21}	a_2	a_3

  - Row 1 contains all a_i symbols. This decomposition has the lossless join property.

- If $R_1 = \{A, B\}$ and $R_2 = \{A, C\}$

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	a_1	b_{22}	a_3

  * For each FDs in F (first loop):
    - Try $AB \rightarrow C$:

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	a_1	b_{22}	a_3

    - Try $C \rightarrow B$:

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	a_1	b_{22}	a_3

    - Complete loop execution results in no changes to the matrix and the matrix does not contain a row with all a_i symbols. This decomposition does not have lossless join property.

11.1.4 Testing Binary Decompositions for the Nonadditive Join Property

- Property LJ1 below is a handy way to decompose a relation into two relations.
  - **Property LJ1** A decomposition $DECOMP = \{R_1, R_2\}$ of R has the lossless join property with respect to a set of functional dependencies F on R *if and only if* either
    * The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+, or
    * The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+.

– Use LJ1 to decompose the previous example.

11.1.5 Successive Lossless (Nonadditive) Join Decompositions

- **Claim 2: Preservarion of Nonadditivity in Successive Decompositions**
  If a decomposition $DECOMP = \{R_1, R_2, \ldots, R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependency F on R, and if a decomposition $D_i = \{Q_1, Q_2, \ldots, Q_k\}$ of R_i has a nonadditive join property with respect to the projection of F on R_i, then the decomposition $D_2 = \{R_1, R_2, \ldots, R_{i-1}, Q_1, Q_2, \ldots, Q_k, R_{i+1}, \ldots, R_m\}$ of R has the nonadditive join property with respect to F.

11.2 Algorithms for Relational Database Schema Design

11.2.1 Dependency-Preserving Decomposition into 3NF Schemas

- **Algorithm 11.2** Relational synthesis algorithm with dependency-preserving
  Input: A universal relation R and a set of functional dependencies F on the attributes of R.
  Output: A dependency-preserving decomposition $DECOMP = \{R_1, R_2, \ldots, R_n\}$ of R that all R_i's in $DECOMP$ are in 3NF.
  - **1.** Find a minimal cover G for F;
  - **2.** For each left-hand-side X of a functional dependency that appears in G, create a relation schema in $DECOMP$ with attributes $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, where $X \rightarrow A_1,\ X \rightarrow A_2,\ \ldots,\ X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the key of this relation);
  - **3.** Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.
- Example:
  $R = \{A, B, C, D, E, H\}$,

$F = \{AE \rightarrow BC,\ B \rightarrow AD,\ CD \rightarrow E,\ E \rightarrow CD,\ A \rightarrow E\}$.

Find a dependency-preserving decomposition $DECOMP = \{R_1, R_2, \ldots, R_n\}$ of R such that each R_i in $DECOMP$ is in 3NF.

- **Step 1:** A minimal cover $G = \{A \rightarrow B,\ A \rightarrow E,\ B \rightarrow A,\ CD \rightarrow E,\ E \rightarrow C,\ E \rightarrow D\}$ of F is derived from algorithm 10.2.
- **Step 2:** Decompose R to

  $R_1 = \{\underline{A}, B, E\}$ and $F_1 = \{A \rightarrow B,\ A \rightarrow E\}$

  $R_2 = \{\underline{B}, A\}$ and $F_2 = \{B \rightarrow A\}$

  $R_3 = \{\underline{C, D}, E\}$ and $F_3 = \{CD \rightarrow E\}$

  $R_4 = \{\underline{E}, C, D\}$ and $F_4 = \{E \rightarrow C,\ E \rightarrow D\}$

  Combine R_3 *and* R_4 into one relation schema

  $R_5 = \{\underline{C, D}, \underline{E}\}$ and $F_5 = \{CD \rightarrow E,\ E \rightarrow C,\ E \rightarrow D\}$.
- **Step 3:** There is one attribute H in $R - (R_1 \cup R_2 \cup R_5)$. Create another relation schema to contain this attribute.

  $R_6 = \{\underline{H}\}$ and $F_6 = \{\}$.

  All relational schemas in the decomposition $DECOMP = \{R_1, R_2, R_5, R_6\}$ are in 3NF.

- Notice that the dependency are preserved: $\{F_1 \cup F_2 \cup F_5 \cup F_6\}^+ = F^+$.
- **Claim 3: Every relation schema created by Algorithm 11.2 is in 3NF.**

11.2.2 Lossless (Nonadditive) Join Decomposition into BCNF Schemas

- **Algorithm 11.3** Relational decomposition into BCNF relations with lossless join preperty

  Input: A universal relation R and a set of functional dependencies F on the attributes of R.

  - **1.** Set $DECOMP = \{R\}$;

- **2.** While there is a relation schema Q in $DECOMP$ that is not in BCNF do
  {
  choose a relation schema Q in $DECOMP$ that is not in BCNF;
  find a functional dependency $X \rightarrow Y$ in Q that violates BCNF;
  replace Q in $DECOMP$ by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
  };

Why: Since $(Q - Y) \cap (X \cup Y) \rightarrow (X \cup Y) - (Q - Y)$ is equvalent to $X \rightarrow Y \ \in F^+$.
By Property LJ1, the decompsition is lossless.

- Example: See Figure 10.11, 10.12, and Figure 10.13 (Fig 14.11, 14.12, 14.13 on e3).

- Example:
  $R = \{A, B, C\}$
  $F = \{AB \rightarrow C,\ C \rightarrow B\}$

  - **Step 1:** Let $DECOMP = \{\{A, B, C\}\}$;
  - **Step 2:** $\{A, B, C\}$ in $DECOMP$ that is not in BCNF;
    Pick $\{A, B, C\}$ in $DECOMP$;
    Pick $C \rightarrow B$ in $\{A, B, C\}$ that violates BCNF;
    Replace $\{A, B, C\}$ in $DECOMP$ by $\{A, C\}$ and $\{B, C\}$;

    Step 2: $DECOMP = \{\{A, C\}, \{B, C\}\}$ and both $\{A, C\}$ and $\{B, C\}$ are in BCNF;
    Therefore, the decomposition $DECOMP = \{\{A, C\}, \{B, C\}\}$ has the lossless join property.
    (Try to use the **Algorithm 11.1** to test this decomposition)

- Another example:
  $R = \{A, B, C, D, E\}$
  $F = \{AB \rightarrow CDE,\ C \rightarrow A,\ E \rightarrow D\}$

  - ‘ **Step 1:** Let $DECOMP = \{\{A, B, C, D, E\}\}$;
  - **Step 2:** $\{A, B, C, D, E\}$ in $DECOMP$ that is not in BCNF;
    Pick $\{A, B, C, D, E\}$ in $DECOMP$;

Pick $C \rightarrow A$ in $\{A, B, C, D, E\}$ that violates BCNF;

Replace $\{A, B, C, D, E\}$ in $DECOMP$ by $\{B, C, D, E\}$ and $\{A, C\}$;

Step 2: $DECOMP = \{\{B, C, D, E\}, \{A, C\}\}$ and $\{B, C, D, E\}$ in $DECOMP$ that is not in

Pick $\{B, C, D, E\}$ in $DECOMP$;

Pick $E \rightarrow D$ in $\{B, C, D, E\}$ that violate BCNF;

Replace $\{B, C, D, E\}$ in $DECOMP$ by $\{B, C, E\}$ and $\{D, E\}$;

Step 2: $DECOMP = \{\{A, C\}, \{B, C, E\}, \{D, E\}\}$ and all of them are in BCNF;

Therefore, the decomposition $DECOMP$ has the lossless join property.

(Try to use the **Algorithm 11.1** to test this decomposition)

- The same example, but try to pick a different FD first that violate BCNF

  $R = \{A, B, C, D, E\}$

  $F = \{AB \rightarrow CDE,\ C \rightarrow A,\ E \rightarrow D\}$

  - **Step 1:** Let $DECOMP = \{\{A, B, C, D, E\}\}$;
  - **Step 2:** $\{A, B, C, D, E\}$ in $DECOMP$ that is not in BCNF;

    Pick $\{A, B, C, D, E\}$ in $DECOMP$;

    Pick $E \rightarrow D$ in $\{A, B, C, D, E\}$ that violates BCNF;

    Replace $\{A, B, C, D, E\}$ in $DECOMP$ by $\{A, B, C, E\}$ and $\{D, E\}$;

    Step 2: $DECOMP = \{\{A, B, C, E\}, \{D, E\}\}$ and $\{A, B, C, E\}$ in $DECOMP$ that is not in

    Pick $\{A, B, C, E\}$ in $DECOMP$;

    Pick $C \rightarrow A$ in $\{A, B, C, E\}$ that violate BCNF;

    Replace $\{A, B, C, E\}$ in $DECOMP$ by $\{B, C, E\}$ and $\{A, C\}$;

    Step 2: $DECOMP = \{\{A, C\}, \{B, C, E\}, \{D, E\}\}$ and all of them are in BCNF;

    Therefore, the decomposition $DECOMP$ has the lossless join property.

    The order of FDs to be applied for decomposition does not matter.

11.2.3 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

- **Algorithm 11.4** Relational synthesis into 3NF with dependency preservation and Nonadditive (lossless) join property

  Input: A universal relation R and a set of functional dependencies F on the attributes of R.

  Output: A dependency-preserving and lossless-join decomposition $DECOMP = \{R_1, R_2, \ldots, R_n\}$ of R that all R_i's in $DECOMP$ are in 3NF.

  - **1.** Find a minimal cover G for F (use algorithm 10.2).
  - **2.** For each left-hand-side X of a functional dependency that appears in G create a relation schema in $DECOMP$ with attributes $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the key of this relation)
  - **3.** If none of the relation schemas in $DECOMP$ contains a key of R, then create one more relation schema in D that contains attributes that form a key of R.

- Example:

  $R = \{A, B, C, D, E, H\}$,

  $F = \{AE \rightarrow BC,\ B \rightarrow AD,\ CD \rightarrow E,\ E \rightarrow CD,\ A \rightarrow E\}$.

  Find a dependency-preserving and lossless-join decomposition $DECOMP = \{R_1, R_2, \ldots, R_n\}$ of R such that each R_i in $DECOMP$ is in 3NF.

  - **Step 1:** A minimal cover $G = \{A \rightarrow B,\ A \rightarrow E,\ B \rightarrow A,\ CD \rightarrow E,\ E \rightarrow CD\}$ of F is derived from algorithm 10.2.
  - **Step 2:** Decompose R to

    $R_1 = \{\underline{A}, B, E\}$ and $F_1 = \{A \rightarrow B,\ A \rightarrow E\}$

    $R_2 = \{\underline{B}, A\}$ and $F_2 = \{B \rightarrow A\}$

    $R_3 = \{\underline{C, D}, E\}$ and $F_3 = \{CD \rightarrow E\}$

    $R_4 = \{\underline{E}, C, D\}$ and $F_4 = \{E \rightarrow CD\}$

    Combine R_3 *and* R_4 into one relation schema

    $R_5 = \{\underline{C, D}, \underline{E}\}$ and $F_5 = \{CD \rightarrow E,\ E \rightarrow CD\}$.
  - **Step 3:** AH and BH are candidate keys of R, and neither of them appear in

R_1, R_2, R_5. Create another relation schema

$R_6 = \{\underline{A, H}\}$ and $F = \{\}$

Then, all relational schemas in the decomposition $DECOMP = \{R_1, R_2, R_5, R_6\}$ are in 3NF.

Dependency preservation: $\{F_1 \cup F_2 \cup F_5 \cup F_6\}^+ = F^+$

Lossless join property: Try to use algorithm 11.1 to test this decomposition.

Appendix: An Example to Summarize Functional Dependencies and Normal Forms

- Let a relation schema $R = \{A, B, C\}$ with functional dependency set

$$F = \{A \rightarrow BC,\ AB \rightarrow C,\ C \rightarrow B\}$$

  - Question#1: Is $G = \{A \rightarrow B,\ AB \rightarrow C,\ C \rightarrow B\}$ equivalent to F ?

    (i) Does $G \models A \rightarrow BC$?

    Since $A_G^+ = ABC \supseteq BC \Rightarrow Yes$.

    (ii) Does $F \models A \rightarrow B$?

    Since $A_F^+ = ABC \supseteq B \Rightarrow Yes$.

    Both (i) and (ii) return true, then $G \equiv F$.

  - Question#2: Find a minimal cover G of F.

    (i) Let $G_1 = \{A \rightarrow B,\ A \rightarrow C,\ AB \rightarrow C,\ C \rightarrow B\}$

    (ii) Check for left redundancy.

    1) Is A redundant in $AB \rightarrow C$ of G_1 ?

    Let $G_2 = \{A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C,\ C \rightarrow B\}$

    Does $G_1 \models B \rightarrow C$? Since $B_{G_1}^+ = B \not\supseteq C \Rightarrow No$

    Does $G_2 \models AB \rightarrow C$?

    Thus, A is **needed**.

    2) Is B redundant in $AB \rightarrow C$ of G_1 ?

Let $G_3 = \{A \rightarrow B,\ A \rightarrow C,\ C \rightarrow B\}$

G_1 **covers** G_3 Since $G_1 \supseteq G_3$

Does $G_3 \models AB \rightarrow C$? Since $AB^+_{G_3} = ABC \supseteq C \Rightarrow Yes.$

Thus, B is **redundant**.

(iii) Check for FDs redundancy.

1) Is $A \rightarrow B$ redundant in G_3 ?

Let $G_4 = \{A \rightarrow C, C \rightarrow B\}$

Does $G_4 \models A \rightarrow B$? $A^+_{G_4} = ACB \supseteq B \Rightarrow Yes.$

Thus, $A \rightarrow B$ is redundant.

2) Is $A \rightarrow C$ redundant in G_4 ?

Let $G_5 = \{C \rightarrow B\}$

Does $G_5 \models A \rightarrow C$? $A^+_{G_5} = A \not\supseteq B \Rightarrow No.$

Thus, $A \rightarrow C$ is **needed**.

Conclusion, $G = G_4 = \{A \rightarrow C,\ C \rightarrow B\}$ is a minimal cover of F.

– Question#3: Find all possible candidate keys of R.

(i) One-attribute keys:

$A^+_F = ABC = R \Rightarrow A$ is a superkey (thus, a candidate key).

$B^+_F = B \neq R$

$C^+_F = CB \neq R$

(ii) Two-attribute keys: We only need to check BC because all other combination will contain A.

$BC^+_F = BC \neq R$

(iii) Three-attribute keys: There is only one possible combination ABC. Since A is a key, ABC is not a key.

Conclusion, There is only one candidate key A.

– Question#4: For the general definition, is R in 2NF ? If not, decompose it to 2NF.

There is only one key with length one. Therefore, no partial dependency exist, i.e., R is in 2NF.

– Question#5: For the general definition, are all relation schemas of the resulting decomposition from Q4 in 3NF ? If not, decompose them to 3NF.

(i) For $A \rightarrow BC$, A is a superkey (ok.)

(ii) For $AB \rightarrow C$, AB is a superkey (ok.)

(iii) For $C \rightarrow B$, C is not a superkey and B is not a prime attribute (not ok.)

Decomposition:

$$R(ABC) \Rightarrow \begin{cases} R_1(\underline{A}C) & F_1 = \{A \rightarrow C\} \\ R_2(B\underline{C}) & F_2 = \{C \rightarrow B\} \end{cases}$$

– Question#6: For the general definition, are all relation schemas of the resulting decompositions from Q5 in BCNF ? If not, decompose them to BCNF.

Both R_1 and R_2 are in BCNF.

– Question#7: Does the resulting decompositions from previous questions have dependency-preserving and lossless-join properties?

(i) Check for lossless-join property.

	A	B	C
R_1	a_1	$\not{b}_{12}$ a_2	a_3
R_2	b_{21}	a_2	a_3

For $A \rightarrow BC$ No change.

$AB \rightarrow C$ No change.

$C \rightarrow B$ change $b_{12} \rightarrow a_2$.

We have a row with all a_i symbols $\Rightarrow$ It has lossless-join property.

(ii) Check for dependency-preserving property.

Does $(F_1 \cup F_2) \equiv F$?

1) Does $F \models A \rightarrow C$? Yes.

2) Does $F_1 \cup F_2 \models A \rightarrow BC,\ AB \rightarrow C$? Yes.

Therefore, it has the dependency-preserving property.

Chapter 14, Indexing Structures for Files

- Data file is stored in secondary memory (disk) due to its large size.
- For accessing a record in data file, it may be necessary to read several blocks from disk to main memory before the correct block is retrieved.
- The index structure provides the more efficient access {fewer blocks access} to records in file based on the **indexing fields** that are used to construct the index.

14.1 Types of Single-level Ordered Indexes

- **Primary index**: specified on the ordering key field of an ordered file of records.
- **Clustering index**: if the ordering field is not a key field.
- **Secondary index**: specified on any nonordering field of a file.

14.1.1 Primary Indexes

- The records of the data file are physically ordered by the primary key.
- A **primary index** is an ordered file whose records are of fixed length with two fields.
- The first field in primary index is of the same data type as the ordering key field (primary key) of the data file.
- The second field in primary index is a pointer to a disk block containing the record.
- Total number of entries in the index is the same as the number of disk blocks in the ordered data file. This is an example of a **nondense index**.
- A binary search on the index file requires fewer block access than a binary search on the data file.
- See Figure 14.1 (Fig 6.1 on e3).
- Example 1:

- r = 30,000 records;
- block size B = 1024 bytes;
- File records are fixed size with record length R = 100 bytes;
- Blocking factor bfr (records per block) = $\lfloor B/R \rfloor$ = 10 records;
- Number of blocks needed is $b = \lceil r/bfr \rceil$ = 3000 blocks;
- A binary search on the data file would need $\lceil log_2 3000 \rceil$ = 12 block accesses;
- Suppose the ordering key field of the file is V = 9 bytes long;
- A block pointer P = 6 bytes long;
- Thus, the size of each index entry R_i in the primary index will be 15 bytes;
- The blocking factor for the index is $bfr_i = \lfloor B/R_i \rfloor$ = 68 entries per block;
- The total index entries r_i is equal to the number of blocks of data file, which is 3000;
- The number of index blocks is hence $b_i = \lceil r_i/bfr_i \rceil$ = 45 blocks.
- The binary search on the index file would need $\lceil log_2 b_i \rceil$ = 6 block accesses.
- To search a record using index, we need one additional block access to the data file for a total $6 + 1 = 7$ block accesses.

- A major problem with a primary index – as with any ordered file – is insertion and deletion of records.

14.1.2 Clustering Indexes

- If records of a file are physically ordered on a nonkey field that does not have a distinct value for each record, that field is called the **clustering field**.
- A clustering index is also an ordered file with two fields.
- The first field is of the same type as the clustering field of the data file.
- The second field is a block pointer.

- There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. See Figure 14.2 and 14.3 (Fig 6.2, 6.3 on e3).
- A clustering index is another example of **nondense index**.
- Insertion and deletion still cause problem, because the data records are physically ordered. Figure 14.3 (Fig 6.3 on e3) shows the way to alleviate the problem)

14.1.3 Secondary Indexes

- A secondary index is also an ordered file with two fields.
- The first field of the index is of the same data type as some *nonordering field* of the data file that is an indexing field.
- The second field of the index is either a block pointer or a record pointer.
- A secondary index can based on a nonordering key field.
  - It is a **dense** index if it is based on nonordering key field, since it contains one entry for each record in the data file.
  - A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. But improvement is much more because the alternative is linear search. See Figure 14.4 (Fig 6.4 on e3).
  - Example 2: (continued from Example 1)
    - Number of records $r = 30{,}000$;
    - Block size $B = 1024$ bytes;
    - A fixed record length $R = 100$ bytes;
    - $bfr = 10$ records per block;
    - Number of blocks of data file = 3000;
    - 1500 average block accesses for a linear search.

    * Suppose we create a secondary index on a nonordering key field.
    * The nonordering key field has 9 bytes long and the block pointer has 6 bytes long;
    * Each entry R_i in the index has 15 bytes long;
    * The blocking factor $bfr_i = \lfloor B/R_i \rfloor = 68$ entries per block;
    * Because the index is a dense index, the total number of entries r_i of the index is equal to $r = 30{,}000$.
    * Thus, the total number of blocks of the index will be $b_i = \lceil r_i/bfr_i \rceil = 442$ blocks.
    * A binary search on the index needs $\lceil log_2 b_i \rceil = 9$ block accesses.
    * 10 block accesses to retrieve a record using the secondary index.

- A secondary index can based on a nonordering nonkey field.
  - **Option 1** is to include several index entries with the same $K(i)$ value – one for each record. This would be a dense index.
  - **Option 2** is to have variable-length records for the index entries, with a repeating field for the pointer.
  - **Option 3** has a fixed length entries and has a single entry for each index field value, but create an extra level of indirection to handle the multiple pointers. Retrieval requires extra block accesses but searching and insertion are straightforward. See Figure 14.5 (Fig 6.5 on e3).

14.1.4 Summary

See Table 14.1 and Table 14.2

14.2 Multilevel Indexes

- The idea here is to reduce the search space by a factor of bfr_i (which is usually >> 2). The blocking factor values bfr_i is called fan out **fo**.
- A multilevel index requires approximately $log_{fo}\ b_i$ block accesses.

- A multilevel index considers the index file, which we will refer to as the **first level** of a multilevel index, as an ordered file with a distinct value of each $K(i)$.
- The **second level** index of a multilevel index is an index to the first level. Since the first level is physically ordered on $K(i)$, the second could have one entry for each block of the first level.
- If the first level has r_1 entries, and the blocking factor – which is also the fan-out – for the index is $bfr_i = fo$, then the first level need $\lceil r_1/fo \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.
- We can repeat the same process until all the entries of some index level t fit in a single block. This block in t^{th} level is call **top** index level.
- $t = \lceil log_{fo}\ (r_1) \rceil$.
- The multilevel index described here can be used on any type of index, whether it is primary, clustering, or secondary – as long as the first level index has distinct values for $K(i)$, fixed-length entries, and is ordered on $K(i)$. See Figure 14.6 (Fig 6.6 on e3).
- Example 3: (continued from Example 2)
  - Suppose that the dense secondary index of example 2 is converted into a multilevel index.
  - The blocking factor is $bfr_i = 68$ entries per block.
  - The number of blocks in the first level is 442 blocks.
  - The number of blocks in the second level will be $\lceil 442/68 \rceil = 7$ blocks.
  - The number of blocks in the third level will be $\lceil 7/68 \rceil = 1$ block.
  - The total block accesses for a search will be $3 + 1 = 4$.
- To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, a **dynamic multilevel index** is implemented by using data structures called $B - tree$ and $B^+ - trees$.

14.3 Dynamic Multilevel Indexes Using B-Tree and B^+-Trees

14.3.1 Search Trees and B-Trees

- A **search tree** is slightly different from a multilevel index.

  A search tree of order p is a tree such that each node contains at most $p-1$ search values and p pointers in the order $< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q >$, where $q \leq p$. All search values are assumed to be unique.

  - Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
  - For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$. See Figure 14.8 (Fig 6.8 on e3).

- In general, we want insertion/deletion algorithms that keep the tree balanced (i.e., all leaf nodes at the same level)

- **B-Trees:** A B-tree of order p when used as an access structure on a key field to search for records in a data file, can be defined as follows. See Figure 14.10 (Fig 6.10 on e3).

  - **Definition:**

    * Each internal node in the B-tree, in Figure 14.10(a) (Fig 6.10(a) on e3), is of the form

      $< P_1, < K_1, Pr_1 >, P_2, < K_2, Pr_2 >, \ldots, < K_{q-1}, Pr_{q-1} >, P_q >$, where $q \leq p$.
    * Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
    * $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
    * Each node has at most p pointers.
    * Each node, except the root node, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
    * A node with q tree pointers, $q \leq p$, has $q-1$ search key field values (and hence has $q-1$ data pointers).

    * All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers P_i are null.

- What do we do if search key is nonkey in data file ?

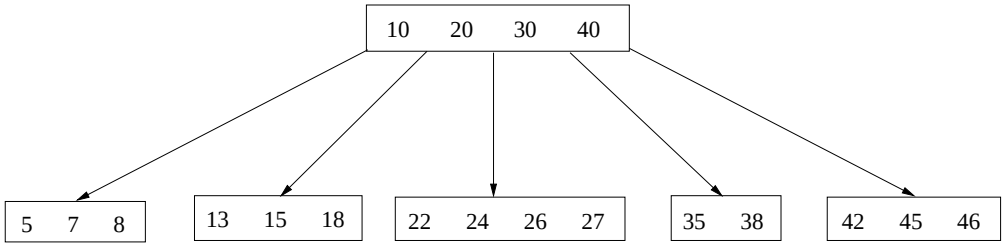
  Keep a linked list of data pointers for each search value.

- **Insertion for B-Tree:** Let p be the order of the B-tree.

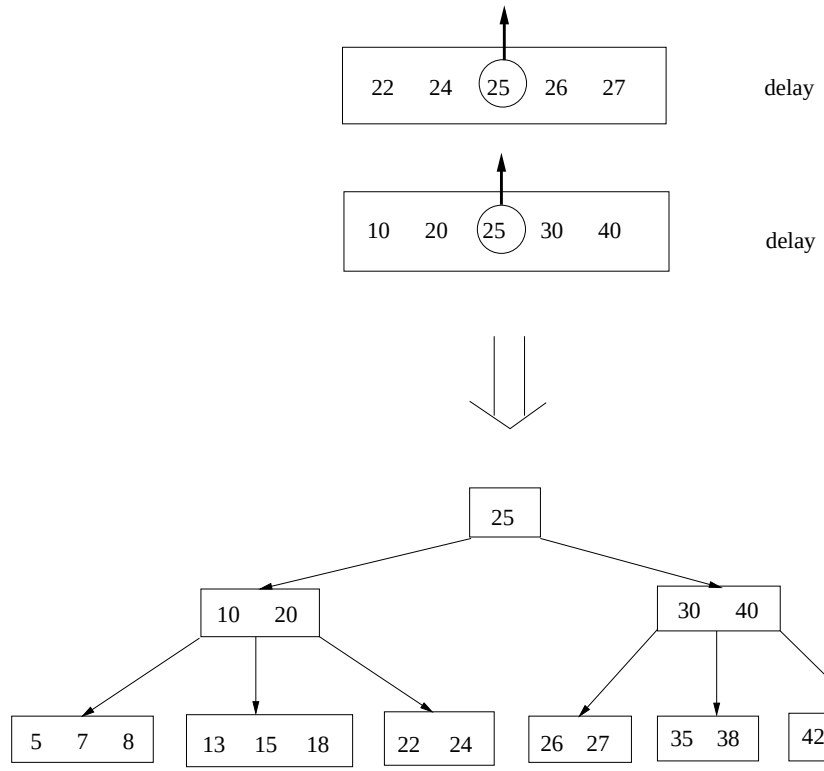
    * B-Tree starts with a single root node at level 0.
    * If any node at level i is full with $p - 1$ search key values and we attempt to insert another value into the node, this node is split into two nodes at the same level i and the middle value goes up to the parent node. If the parent node is full, it is also split. This can propagate all the way to the root, creating a new root if the old root is split.

– Example for B-Tree insertions:

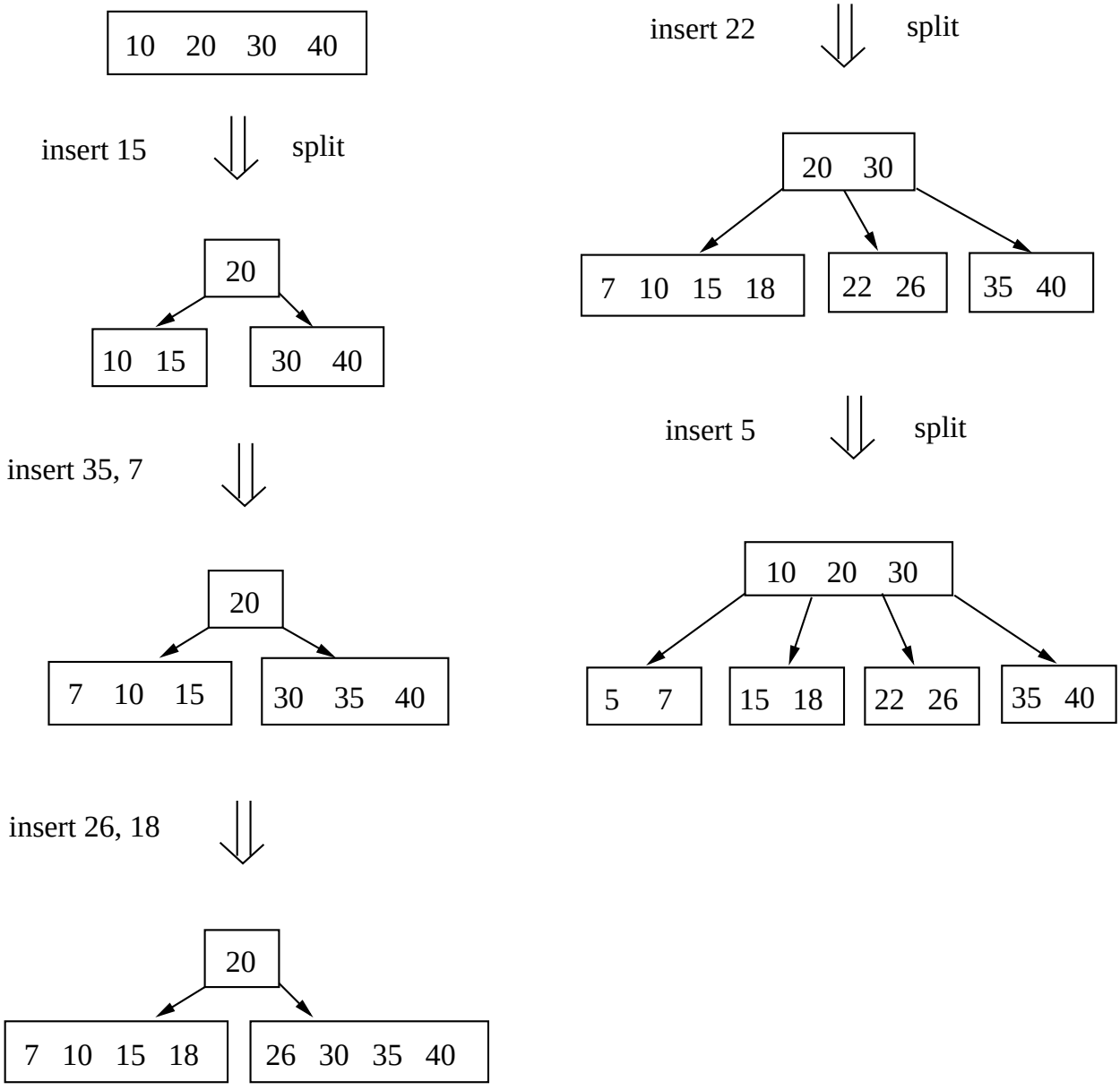
  * If exists a B-Tree with $p = 5$ as below. What happened if we insert 25 ?



If insert 25, what happened ?

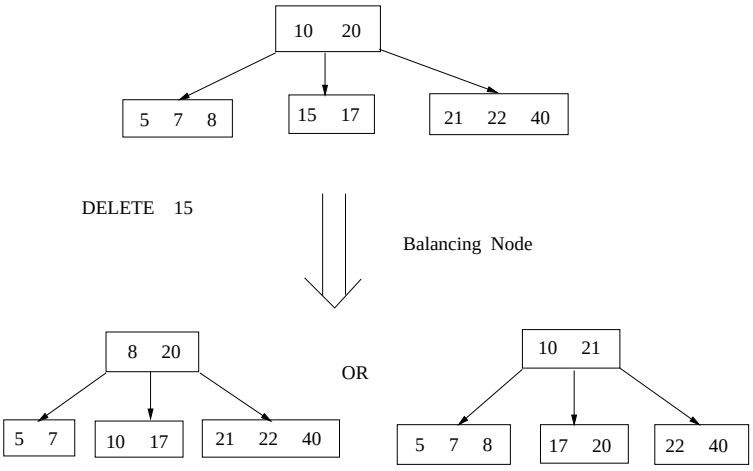


$*$ Let $p = 5$; input: 20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5.

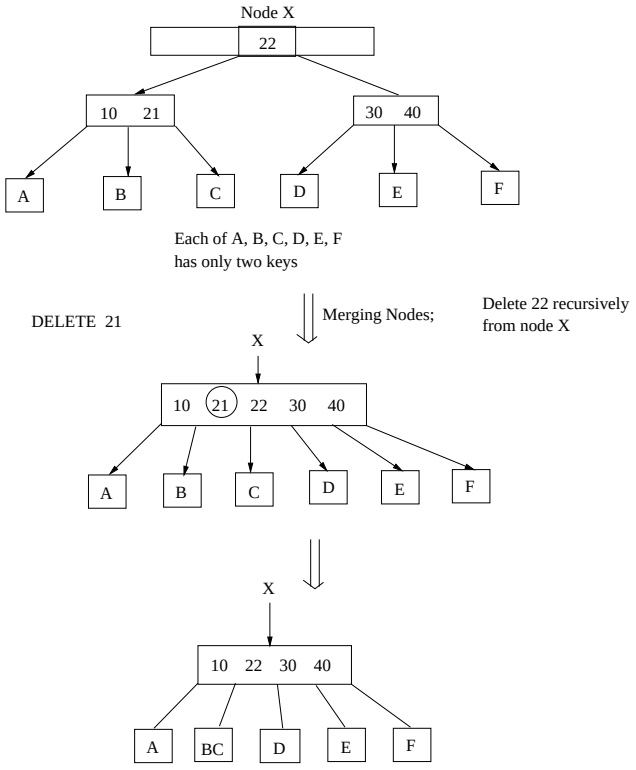


– **Deletion for B-Tree:** Dealing with underflow.

* **Balancing nodes:** If there are extra keys in its left or right subtrees (left or right siblings if the value being deleted is in the leaf node). Let $p = 5$.

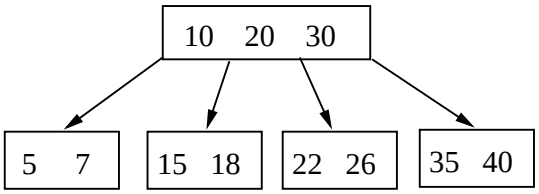


* **Merging nodes:** If there are no extra keys in its left and right subtrees. Let $p = 5$.

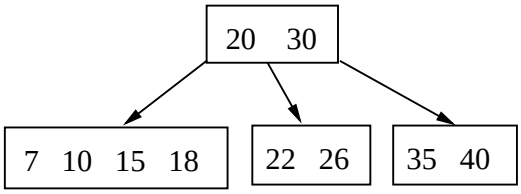


– Example for B-Tree deletions:

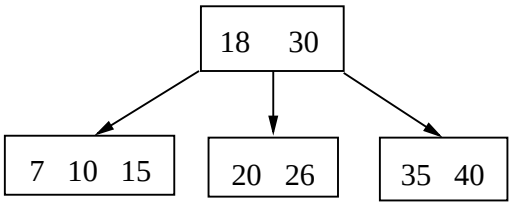
∗ Let $p = 5$; delete 5, 22, 18, 26, 7, 35, 15 from the B-Tree below.



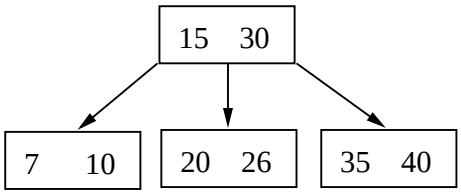
delete 5 ⇓ merging



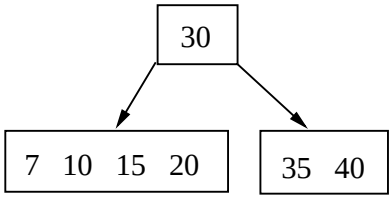
delete 22 ⇓ balancing



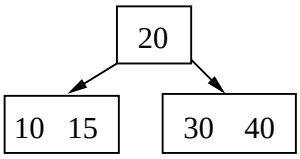
delete 18 ⇓ balancing



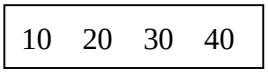
delete 26 ⇓ merging



delete 7, 35 ⇓ balancing



delete 15 ⇓ merging



- Example 4: How to choose p for B-Trees.
  - Suppose the search field is $V = 9$ bytes long.
  - Suppose the disk block size is $B = 512$ bytes.
  - Suppose a record (data) pointer is $P_r = 7$ bytes.
  - Suppose a block pointer is $P = 6$ bytes.
  - Each B-Tree nodes can have at most p tree pointers, $p-1$ data pointers, and $p-1$ search key field values.
  - Suppose each B-Tree node is to correspond to a disk block. Then we have:

$$(p \times P) + ((p-1) \times (P_r + V)) \leq B$$

$$(p \times 6) + ((p-1) \times (7+9)) \leq 512$$

$$(22 \times p) \leq 512$$

    Then, we can choose $p = 23$ (not 24, because we want to have extra space for addition information)

- Example 5: examine the capacity of the B-Tree.
  - Suppose that the search field of Example 4 is a nonordering key field, and we construct a B-Tree on this field.
  - Assume that each node of the B-Tree is 69 percent full.
  - Each node, on the average, will have $p \times 0.69 = 23 \times 0.69$ or approximately 16 pointers and, hence, 15 search key field values.
  - The average fan-out $fo = 16$. Then

Root:	1 nodes	15 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3840 entries	4096 pointers
Level 3:	4096 nodes	61,440 entries	
Total		65,535 entries	

- B-Tree can also be used as primary file organizations. In this case, whole records are stored within the B-Tree nodes rather than the search key and pointers.

14.3.2 B^+-Trees

- Every value of the search field appears once at some level in a B-Tree.
- In a B^+-Tree, data pointers are stored only at the leaf nodes; hence, the structure of leaf nodes differs from the structure of internal nodes.
- A leaf node entry: < search value, data pointer >.
- An internal node entry: < search value, tree pointer >.
- Leaf nodes are usually linked together to provide ordered access on the search field to the records.
- Some search field values from the leaf nodes are repeated in the internal nodes to guide the search.
- **Definition:**
  - The structure of **internal nodes:**
    - Each internal node is of the form

      $$< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q >$$

      Where $q \leq p$ and each P_i is a tree pointer.
    - Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.
    - For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$, See Figure 14.11(a) (Fig 6.11(a) on e3).
    - Each internal node has at most p tree pointers.
    - Each internal node, except the root, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
    - An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.
  - The structure of **leaf nodes:**

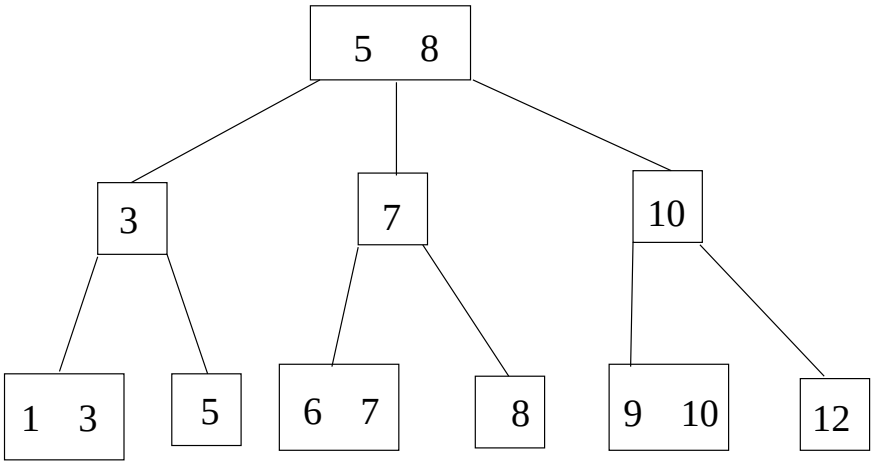
* Each leaf node is of the form

$$<< K_1, Pr_1 >, < K_2, Pr_2 >, \ldots, < K_{q-1}, Pr_{q-1} >, P_{next} >$$

where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next leaf node of the B$^+$-Tree.

* Within each leaf node, $K_1 < K_2 < \ldots < K_{q-1}$, where $q \leq p$.
* Each Pr_i is a data pointer that points to the record whose search field value is K_i or to a file block containing the record.
* Each leaf node has at least $\lceil p/2 \rceil$ values.
* all leaf nodes are at the same level.

- Example of B$^+$-Tree with $p = 3$ for internal nodes and $p_{leaf} = 2$ for leaf nodes.



- Every value appearing in an internal node also appears as the *rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

- Every key value must exist at the leaf level, because all data pointers are at the leaf level. Only some values exist in internal nodes to guide the search.

- Example 6: To calculate the orders p and p_{leaf} of a B$^+$-Tree.
  - Search key field $V = 9$ bytes.
  - The block size $B = 512$ bytes.

- A record pointer $Pr = 7$ bytes.
- A tree pointer $P = 6$ bytes.
- p tree pointers and $p - 1$ search values in an internal node (block)

$$(p \times P) + ((p - 1) \times V) \leq B$$

$$(p \times 6) + ((p - 1) \times 9) \leq 512$$

We choose $p = 34$.

- p_{leaf} data pointers and P_{leaf} search values in a leaf node (block)

$$(p_{leaf} \times (Pr + V)) + P \leq B$$

$$(p_{leaf} \times (7 + 9)) + 6 \leq 512$$

we choose $p_{leaf} = 31$.

- Example 7: examine the capacity of the B^+-Tree.
  - Assume that each node of the B^+-Tree is 69 percent full.
  - Each internal node, on the average, will have $p \times 0.69 = 34 \times 0.69$ or approximately 23 pointers and, hence, 22 search key field values.
  - Each leaf node, on the average, will have $p_{leaf} \times 0.69 = 31 \times 0.69$ or approximately 21 data record pointers and, hence, 21 search key field values.
  - The B^+-Tree will have the following average number of entries at each level.

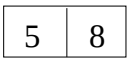
Root:	1 nodes	22 entries	23 pointers
Level 1:	23 nodes	506 entries	529 pointers
Level 2:	529 nodes	11,638 entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 entries	

  - Compare to B-Tree (in example 5) which has 65,535 entries.
- **Insertion** for B^+-Trees.
  - **Splitting a leaf node because of overflow:** When a leaf node is full and a new value is inserted there, the node **overflows** and must be split. The first

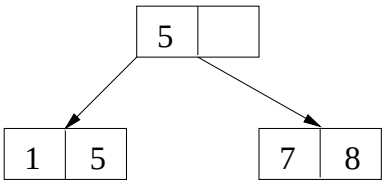
$j = \lceil (P_{leaf} + 1)/2 \rceil$ entries are kept in the original node, and the remaining entries are moved to a new leaf node. The j^{th} value is replicated in the parent internal node.

- If the parent internal is full, the new value (the j^{th} value in above step) will cause it to overflow also, so it must be split.
- **Splitting an internal node because of overflow:** the entries in the internal node up to P_j – the j^{th} tree pointer after inserting the new value and pointer, where $j = \lfloor (p + 1)/2 \rfloor$ – are kept, while the j^{th} search value in moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node.
- This splitting can propagate all the way up to create a new root and hence a new level for the B$^+$-Tree.

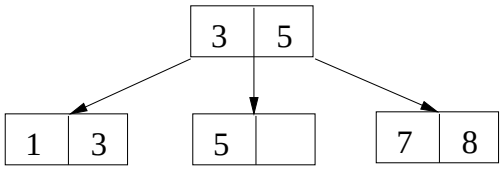
- Example of the B$^+$-Tree insertion: Let a B$^+$-Tree with $p = 3$ and $p_{leaf} = 2$. Try to create the B$^+$-Tree by inserting the following sequence of numbers – 8, 5, 1, 7, 3, 12, 9, 6 (See next page).



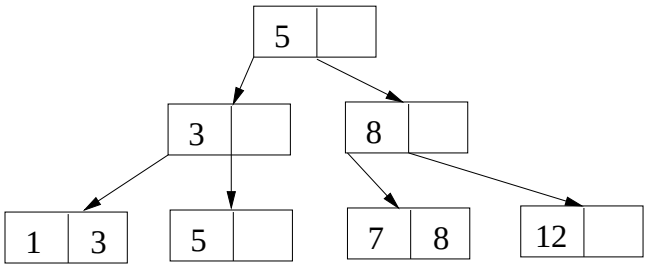
After inserting 8, 5



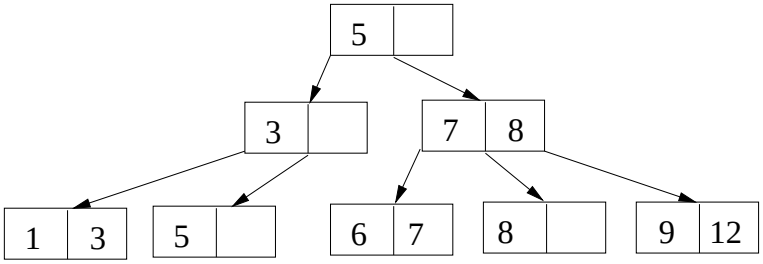
After inserting 1, 7



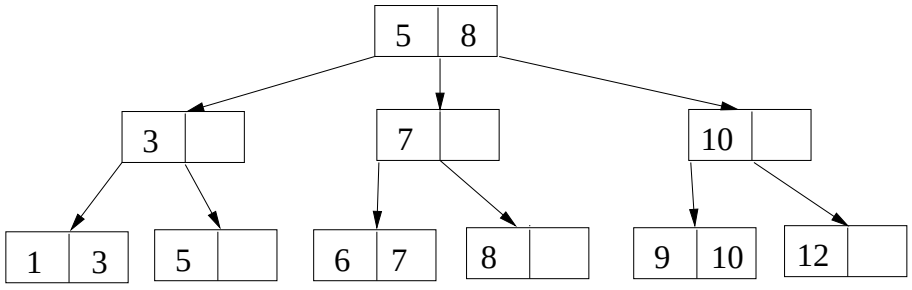
After inserting 3



After inserting 12



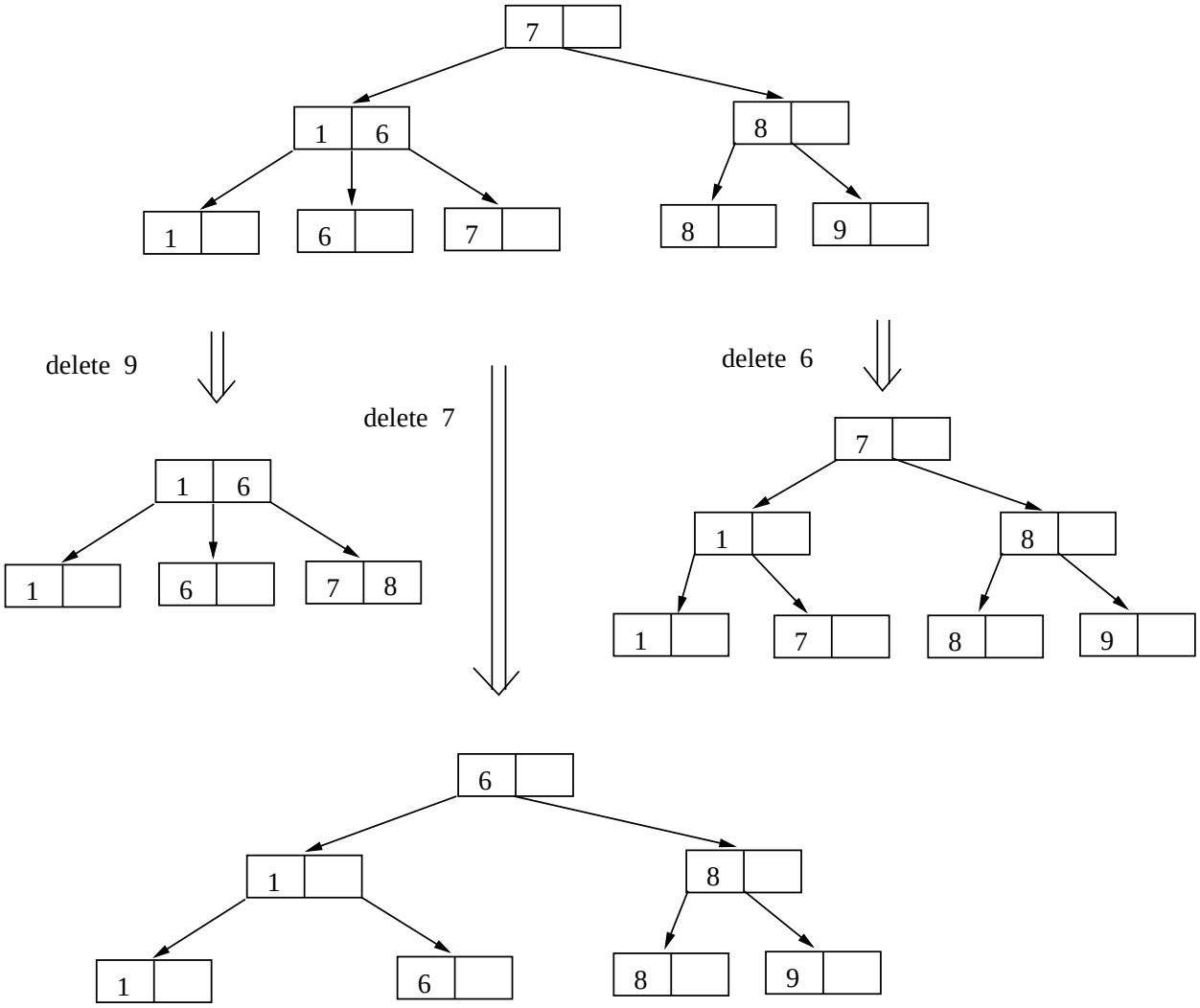
After inserting 9, 6



After inserting 10

- **Deletion** for B^+-Trees:
  - When an entry is deleted, it is always removed from the leaf node. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node.
  - **Underflow in the leaf node:**
    * Try to **redistribute (balancing)** the entries from the left sibling if possible.
    * Try to **redistribute (balancing)** the entries from the right sibling if possible.
    * If the redistribution is not possible, then the three nodes are merged into two leaf nodes. In this case, underflow may propagate to internal nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.
- Example of the B^+-Tree deletion:
  - see Figure 14.13 (Fig 6.13 on e3).

– Let $p = 3$ and $p_{leaf} = 2$; delete 9 or 6 or 7 from a B$^+$-Tree below.



Chapter 15, Algorithms for Query Processing and Optimization

- A query expressed in a high-level query language such as SQL must be **scanned, parsed, and validate**.
- Scanner: identify the language tokens.
- Parser: check query syntax.
- Validate: check all attribute and relation names are valid.
- An internal representation (**query tree or query graph**) of the query is created after scanning, parsing, and validating.
- Then DBMS must devise an **execution strategy** for retrieving the result from the database files.
- How to choose a suitable (efficient) strategy for processing a query is known as **query optimization**.
- The term optimization is actually a misnomer because in some cases the chosen execution plan is not the optimal strategy – it is just a reasonably efficient one.
- There are two main techniques for implementing query optimization.
  - **Heuristic rules** for re-ordering the operations in a query.
  - **Systematically estimating** the cost of different execution strategies and choosing the lowest cost estimate.

15.1 Translating SQL Queries into Relation Algebra

- An SQL query is first translated into an equivalent extended relation algebra expression (as a query tree) that is then optimized.

- **Query block** in SQL: the basic unit that can be translated into the algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses.
  - Consider the following SQL query.

    SELECT LNAME, FNAME
    FROM EMPLOYEE
    WHERE SALARY > (**SELECT** **MAX** (SALARY)
    FROM EMPLOYEE
    WHERE DNO=5);

  - We can decompose it into two blocks:

    Inner block:
    (**SELECT** **MAX** (SALARY)
    FROM EMPLOYEE
    WHERE DNO=5)

    Outer block:
    SELECT LNAME, FNAME
    FROM EMPLOYEE
    WHERE SALARY > **c**

  - Then translate to algebra expressions:
    * Inner block: $\Im_{MAX\ SALARY}(\sigma_{DNO=5}\ EMPLOYEE)$
    * Outer block: $\pi_{LNAME,\ FNAME}(\sigma_{SALARY>c}\ EMPLOYEE)$

15.2 Algorithms for External Sorting

- **External sorting** refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory.
- Use a **sort-merge strategy**, which starts by sorting small subfiles – called **runs** – of the main file and merges the sorted runs, creating larger sorted subfiles that are merged in turn.
- The algorithm consists of two phases: sorting phase and merging phase.
- **Sorting phase:**

- Runs of the file that can fit in the available buffer space are read into main memory, sorted using an internal sorting algorithm, and written back to disk as temporary sorted subfiles (or runs).
- **number of initial runs (n_R), number of file blocks (b), and available buffer space (n_b)**
- $n_R = \lceil b/n_B \rceil$
- If the available buffer size is 5 blocks and the file contains 1024 blocks, then there are 205 initial runs each of size 5 blocks. After the sort phase, 205 sorted runs are stored as temporary subfiles on disk.

- **Merging phase:**
  - The sorted runs are merged during one or more **passes**.
  - The **degree of merging (d_M)** is the number of runs that can be merged together in each pass.
  - In each pass, one buffer block is needed to hold one block from each of the runs being merged, and one block is needed for containing one block of the merge result.
  - $d_M = MIN\{n_B - 1, n_R\}$, and the number of passes is $\lceil log_{d_M}(n_R) \rceil$.
  - In previous example, $d_M = 4$, 205 runs $\rightarrow$ 52 runs $\rightarrow$ 13 runs $\rightarrow$ 4 runs $\rightarrow$ 1 run. This means 4 passes.

- The complexity of external sorting (number of block accesses): $(2 \times b) + (2 \times (b \times (log_{d_M}\ b)))$

- For example:
  - 5 initial runs $[2, 8, 11], [4, 6, 7], [1, 9, 13], [3, 12, 15], [5, 10, 14]$.
  - The available buffer $n_B = 3$ blocks $\rightarrow d_M = 2$ (two way merge)
  - After first pass: 3 runs
    $[\underline{2, 4, 6}, \underline{7, 8, 11}]$, $[\underline{1, 3, 9}, \underline{12, 13, 15}]$, $[\underline{5, 10, 14}]$

- After second pass: 2 runs

  $[\underline{1,2,3}, \underline{4,6,7}, \underline{8,9,11}, \underline{12,13,15}]$, $[\underline{5,10,14}]$

- After third pass:

  $[\underline{1,2,3}, \underline{4,5,6}, \underline{7,8,9}, \underline{10,11,12}, \underline{13,14,15}]$

15.3 Algorithms for SELECT and JOIN Operations

15.3.1 Implementing the SELECT Operation

- Five operations for demonstration:
  - (OP1): $\sigma_{SSN='123456789'}\ (EMPLOYEE)$
  - (OP2): $\sigma_{DNUMBER>5}\ (DEPARTMENT)$
  - (OP3): $\sigma_{DNO=5}\ (EMPLOYEE)$
  - (OP4): $\sigma_{DNO=5\ and\ SALARY>30000\ and\ SEX='F'}\ (EMPLOYEE)$
  - (OP5): $\sigma_{ESSN='123456789'\ and\ PNO=10}\ (WORKS\_ON)$
- **Search methods for simple selection:**
  - **S1.** *Linear search (brute force)*
  - **S2.** *Binary search*: If the selection condition involves an equality comparison on a key attribute on which the file is ordered. (ex. OP1)
  - **S3.** *Using a primary index*: If the selection condition involves an equality comparison on a key attribute with a primary index. (ex. OP1)
  - **S4.** *Using a primary index to retrieve multiple records*: If the comparison condition is $>, \geq, <, \geq$ on a key field with a primary index – for example, $DNUMBER > 5$ in OP2 – use the index to find the record satisfying the equality condition ($DNUMBER = 5$), then retrieve all subsequent (preceding) records in the (ordered) file.

- **S5.** *Using a clustering index to retrieve multiple records*: If the selection condition involves an equality comparison on a non-key attribute with a clustering index – for example, $DNO = 5$ in OP3 – use the index to retrieve all the records satisfying the condition.
- **S6.** *Using a secondary (B^+-Tree) index on an equality comparison*: Retrieve one record if the indexing field is a key, or multiple records if the indexing field is not a key. This method can also be used for comparison involved $>, \leq, <, \geq$.

- **Search methods for complex (conjunctive) selection:**
  - **S7.** *Conjunctive selection using an individual index*: If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6 to retrieve the records and then check whether each retrieved record satisfies the remaining conditions in the conjunctive condition.
  - **S8.** *Conjunctive selection using a composite index*: If two or more attributes are involved in equality conditions and a composite index exists on the combined fields, we can use the index directly. For example, OP5 can use this method if there is a composite index $(ESSN, PNO)$ for WORKS_ON file.
  - **S9.** *Conjunctive selection by intersection of record pointers*: If secondary indexes are available on more than one of the fields in the conjunctive condition, and if the indexes include record pointers (rather than block pointers). The intersection of these sets of record pointers would satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the simple conditions have secondary indexes, each retrieved record is further tested for the remaining conditions.

15.3.2 Implementing the JOIN Operation

- Two operations for demonstration:

- (OP6): $EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT$
- (OP7): $DEPARTMENT \bowtie_{MGRSSN=SSN} EMPLOYEE$

- **Methods for implementing JOINs:**

  The algorithm we consider are for join operation of the form

$$R \bowtie_{A=B} S$$

  - **J1.** *Nested-loop join (brute force)*: For each record t in R (outer loop), retrieve every record s from S and test whether the two records satisfy the join condition $t[A] = s[B]$.
  - **J2.** *Single-loop join (using an access structure to retrieve the matching records)*: If an index exists for one of the two attributes – say, B of S – retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.
  - **J3.** *Sort-merge join*: First to sort R and S based on the corresponding attributes A and B using external sorting. Second to merge sorted R and S by retrieving the matching records t and s that satisfy the join condition $t[A] = s[B]$.

    Note that if there are secondary indexes for R and S based on attributes A and B, then we can merge these two secondary indexes instead of sorting and merging the data files R and S.
  - **J4.** *Hash-join*:
    - **Partitioning phase:** a single pass through the file with fewer records (say, R) hashes its records (using A as the hash key) to the hash file buckets
    - **Probing phase:** a single pass through the other file (S) then hashes each of its records to probe the appropriate bucket.

15.4 Algorithms for PROJECT and SET Operations

- **PROJECT operation:**

- If the $<$ *attribute list* $>$ of the PROJECT operation $\pi_{<attribute\ list>}(R)$ includes a key of R, then the number of tuples in the projection result is equal to the number of tuples in R, but only with the values for the attributes in $<$ *attribute list* $>$ in each tuple.
- If the $<$ *attribute list* $>$ does not contain a key of R, duplicate tuples must be eliminated. The following methods can be used to eliminate duplication.
  * Sorting the result of the operation and then eliminating duplicate tuples.
  * Hashing the result of the operation into hash file in memory and check each hashed record against those in the same bucket; if it is a duplicate, it is not inserted.

- **CARTESIAN PRODUCT operation:**
  - It is an extremely expensive operation. Hence, it is important to avoid this operation and to substitute other equivalent operations during optimization.

- **UNION, INTERSECTION, and DIFFERENCE operations:**
  - Use variations of the **sort-merge** technique.
  - Use Hashing technique. That is, first hash (partition) one file, then hash (probe) another file.

15.5 Implementing Aggregate Operations and OUTER JOINS

15.5.1 Implementing Aggregate Operations

- The aggregate operators (MIN, MAX, COUNT, AVERAGE, SUM), when applied to an entire table, can be computed by a **table scan** or by using an appropriate index.

- **SELECT** **MAX**(SALARY)
  FROM EMPLOYEE;

If an (ascending) index on SALARY exists for the EMPLOYEE relation, the optimizer can decide the largest value: the rightmost leaf (B-Tree and B^+-Tree) or the last entry in the first level index (clustering, secondary).

- The index could also be used for COUNT, AVERAGE, and SUM aggregates, if it is a **dense index**. For a **nondense index**, the actual number of records associated with each index entry must be used for a correct computation.
- For a GROUP BY clause in a query, the technique to use is to partition (either **sorting** or **hashing**) the relation on the grouping attributes and then to apply the aggregate operators on each group.
  - If a clustering index exists on the grouping attributes, then the records are already partitioned.

15.5.2 Implementing Outer Join

- Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join.
  - For a left (right) outer join, we use the left (right) relation as the outer loop or single-loop in the join algorithms.
  - If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with null values.
- The other join algorithms, **sort-merge and hash-join**, can also be extended to compute outer joins.

15.6 Combining Operations Using Pipelining

- A query is typically a sequence of relational operations. If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead.

- **Pipelining** processing: Instead of generating temporary files on disk, the result tuples from one operation are provided directly as input for subsequent operations.

15.7 Using Heuristics in Query Optimization

- Using heuristic rules to modify the internal representation (query tree) of a query.
- One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operations.
- The SELECT and PROJECT operations reduce the size of a file and hence should be applied first.

15.7.1 Notation for Query Trees and Query Graphs

- **Query tree:** see Figure 15.4(a) (Fig 18.4(a) on e3).
  - A tree data structure that corresponds to a relational algebra expression. The input relations of the query – leaf nodes; the relational algebra operations – internal nodes.
  - An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
  - A query tree specifies a specific order of operations for executing a query.
- **Query Graph:** see Figure 15.4(c) (Fig 18.4(c) on e3).
  - **Relation nodes:** displayed as single circles.
  - **Constant nodes:** displayed as double circles.
  - **Graph edges:** specify the selection and join conditions.
  - The attributes to be retrieved from each relation are displayed in square brackets above each relation.

- The query graph does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query. Hence, a query graph corresponds to a *relational calculus* expression.

15.7.2 Heuristic Optimization of Query Trees

- Many different relational algebra expressions – and hence many different query trees – can be equivalent.
- The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization.
- The **initial query tree (canonical query tree)** is generated by the following sequence.
  - The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied.
  - The selection and join conditions of the WHERE clause are applied.
  - The projection on the SELECT clause attributes are applied.
- The heuristic query optimizer transform this initial query tree (inefficient) to a final query tree that is efficient to execute.
- Example of transforming a query. See Figure 15.5 (Fig 18.5 on e3).
  Consider the SQL query below.

```
SELECT  LNAME
FROM    EMPLOYEE, WORKS_ON, PROJECT
WHERE   PNAME='Aquarius' and PNUMBER=PNO and ESSN=SSN
        and BDATE > '1957-12-31';
```

- **General transformation rules for relational Algebra operations:**

– **1. Cascade of σ:**

$\sigma_{\ c_1 \ and \ c_2 \ and \ \ldots \ and \ c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

– **2. Commutativity of σ:**

$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$

– **3. Cascade of π:**

$\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) \equiv \pi_{List1}(R)$

– **4. Commuting σ with π:** If the selection condition c involves only those attributes $A_1, A_2, \ldots, A_n$ in the projection list.

$\pi_{A_1,A_2,\ldots,A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,A_2,\ldots,A_n}(R))$

– **5. Commutativity of $\bowtie$ (and $\times$):**

$R \bowtie_c S \equiv S \bowtie_c R$

$R \times S \equiv S \times R$

– **6. Commuting σ with $\bowtie$ (or $\times$):** If the selection condition c can be written as (c_1 *and* c_2), where c_1 involves only the attributes of R and c_2 involves only the attributes of S.

$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$

– **7. Commuting π with $\bowtie$ (or $\times$):** Suppose the projection list $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S

  * If the join condition c involves only attributes in L.

    $\pi_L(R \bowtie_c S) \equiv (\pi_{A_1,\ldots,A_n}(R)) \bowtie_c (\pi_{B_1,\ldots,B_m}(S))$

  * If the join condition c contains additional attributes not in L. For example, $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S.

    $\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A_1,\ldots,A_n,A_{n+1},\ldots,A_{n+k}}(R)) \bowtie_c (\pi_{B_1,\ldots,B_m,B_{m+1},\ldots,B_{m+p}}(S)))$

– **8. Commutativity of set operations:** $\cap$ and $\cup$ are commutative, but not $-$.

– **9. Associativity of $\bowtie, \times, \cap, \cup$:** Let θ be one of the four operations.

$(R \ \theta \ S) \ \theta \ T \equiv R \ \theta \ (S \ \theta \ T)$

- **10. Commuting σ with set operations:** Let θ be one of the three set operations $\cap$, $\cup$, and $-$.

  $\sigma_c(R\ \theta\ S) \equiv (\sigma_c(R))\ \theta\ (\sigma_c(S))$

- **11. The π operation commutes with $\cup$:**

  $\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$

- **12. Converting a $(\sigma, \times)$ sequence into $\bowtie$:**

  $(\sigma_c(R \times S)) \equiv (R \bowtie_c S)$

- Another possible transformations such as DeMorgan's law:

  $not\ (c_1\ and\ c_2) \equiv (not\ c_1)\ or\ (not\ c_2)$

  $not\ (c_1\ or\ c_2) \equiv (not\ c_1)\ and\ (not\ c_2)$

- **Outline of a heuristic algebraic optimization algorithm:**

  - **1. Break up the SELECT operations:**

    Using rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations.

  - **2. Push down the SELECT operations:**

    Using rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the tree as is permitted by the attributes involved in the select condition.

  - **3. Rearrange the leaf nodes:**

    Using rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria.

    * 1) Position the leaf node relations with most restrictive SELECT operations so they are executed first in the query tree.
    * 2) Make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations.

  - **4. Change CARTESIAN PRODUCT to JOIN operations:**

    Using rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation.

- **5. Break up and push down PROJECT operations:**

  Using rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed.

- **6. Identify subtrees for pipelining:**

  Identify subtrees that represent groups of operations that can be executed by a single algorithm.

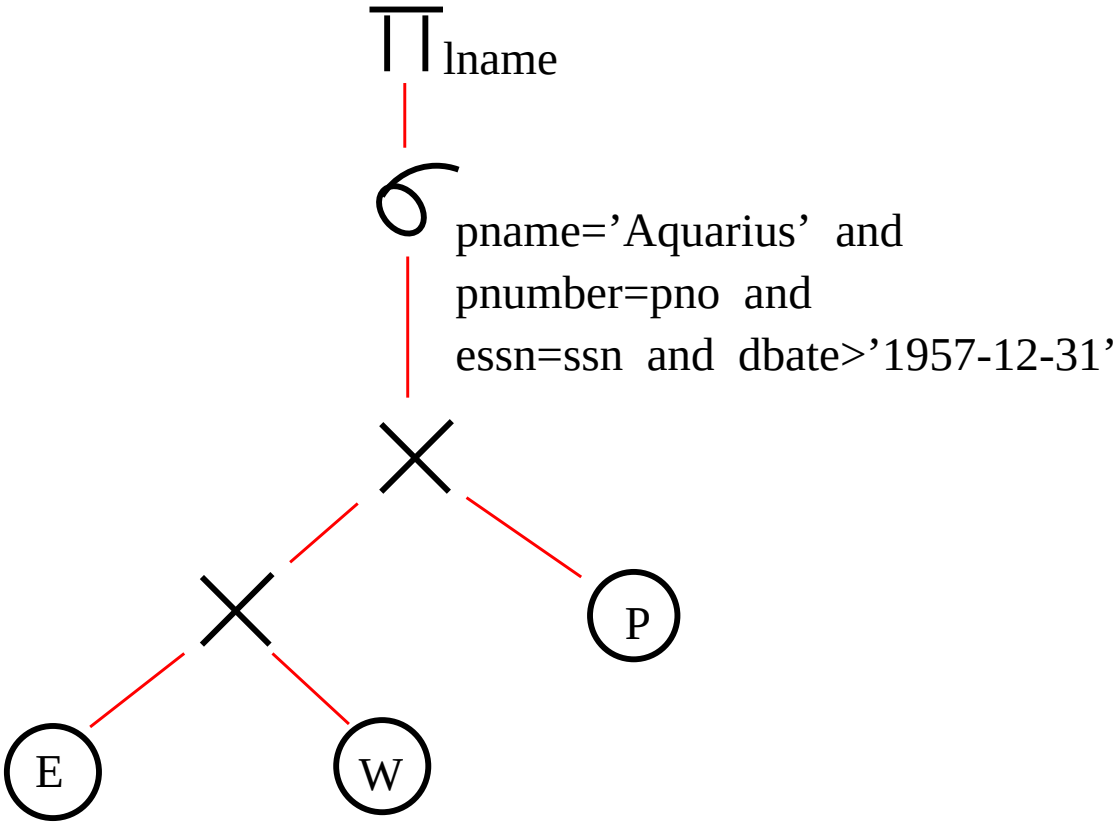
- Example for transforming SQL query ⇒ Initial query tree ⇒ Optimized query tree.

  SELECT lname
  FROM employee, works_on, project
  WHERE pname='Aquarius' **and** pnumber=pno **and** essn=ssn
  and bdate > '1957-12-31';

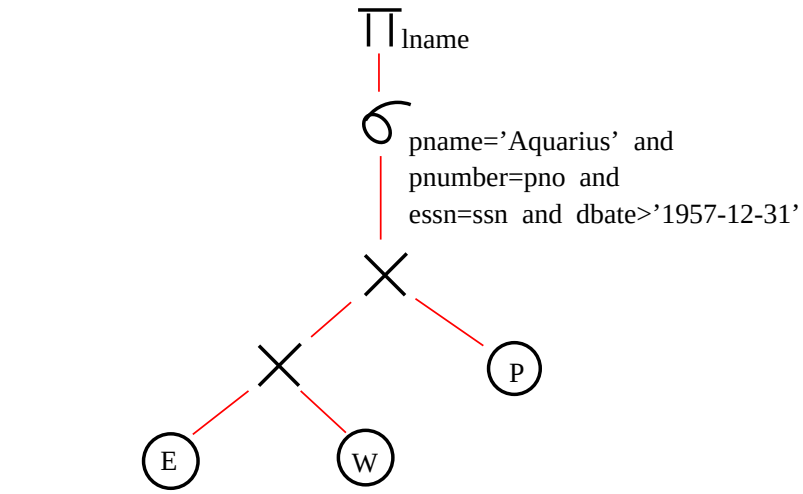
  Suppose the selection cardinalities (number of tuples in the resulting relation after applying the selection operation) for all selection conditions in the above WHERE clause are as follows.

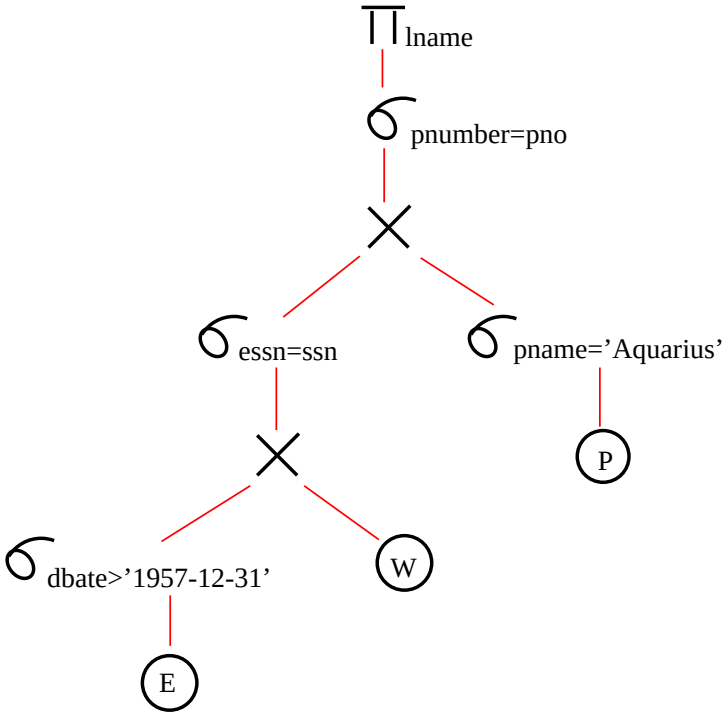
	$\sigma_{pname='Aquarius'}$	$\sigma_{bdate>'1957-12-31'}$
relation	project	employee
cardinality	1	3

– SQL query ⇒ initial query tree.

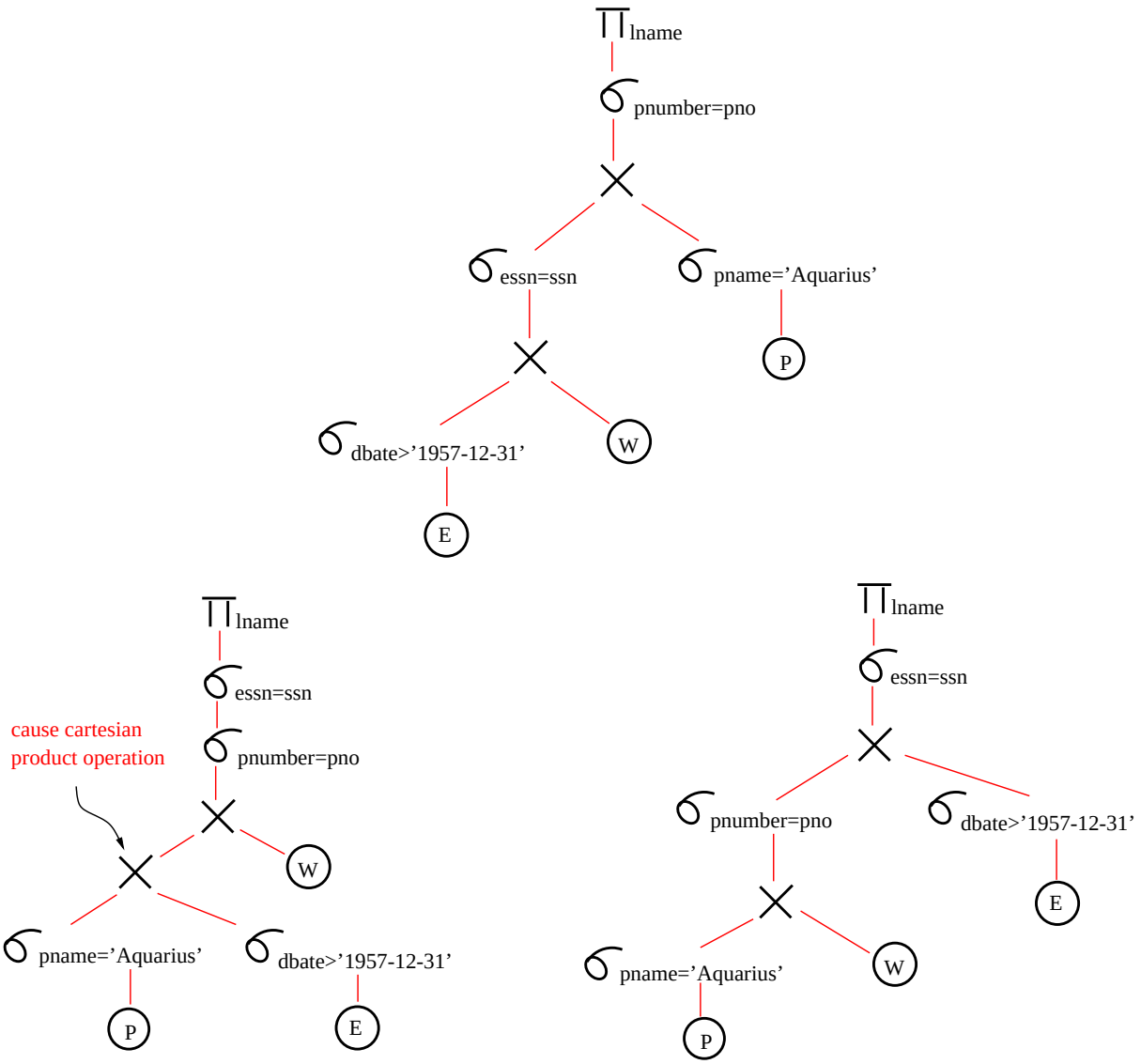


– initial query tree ⇒ query tree after pushing down selection operation.

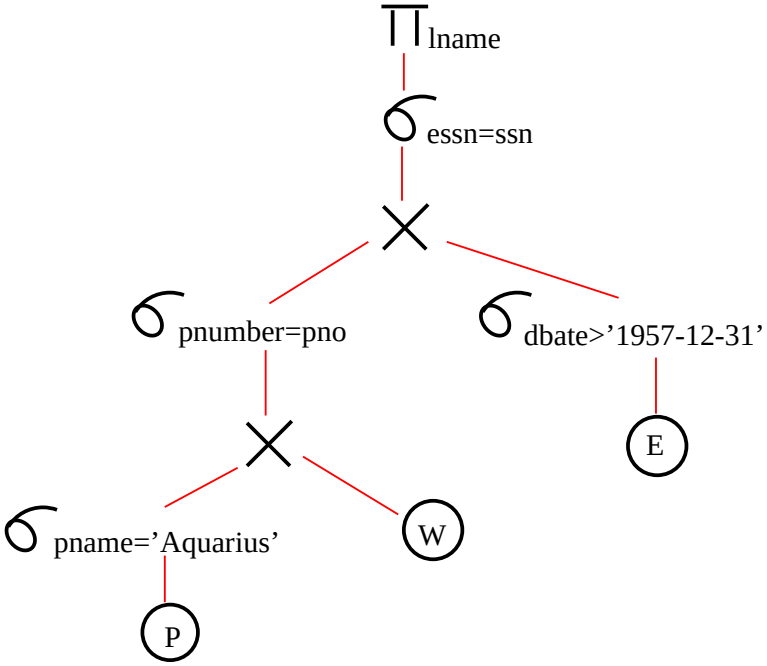


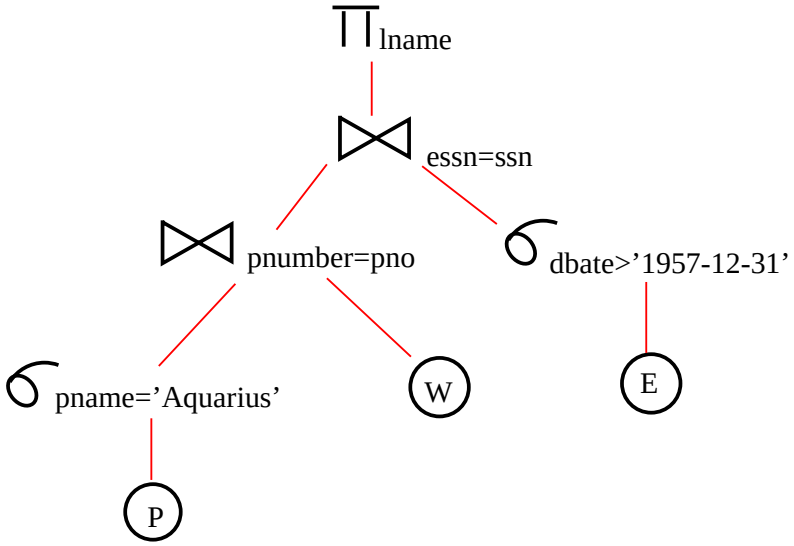


– query tree after pushing down selection ⇒ query tree after leaf nodes re-ordering.

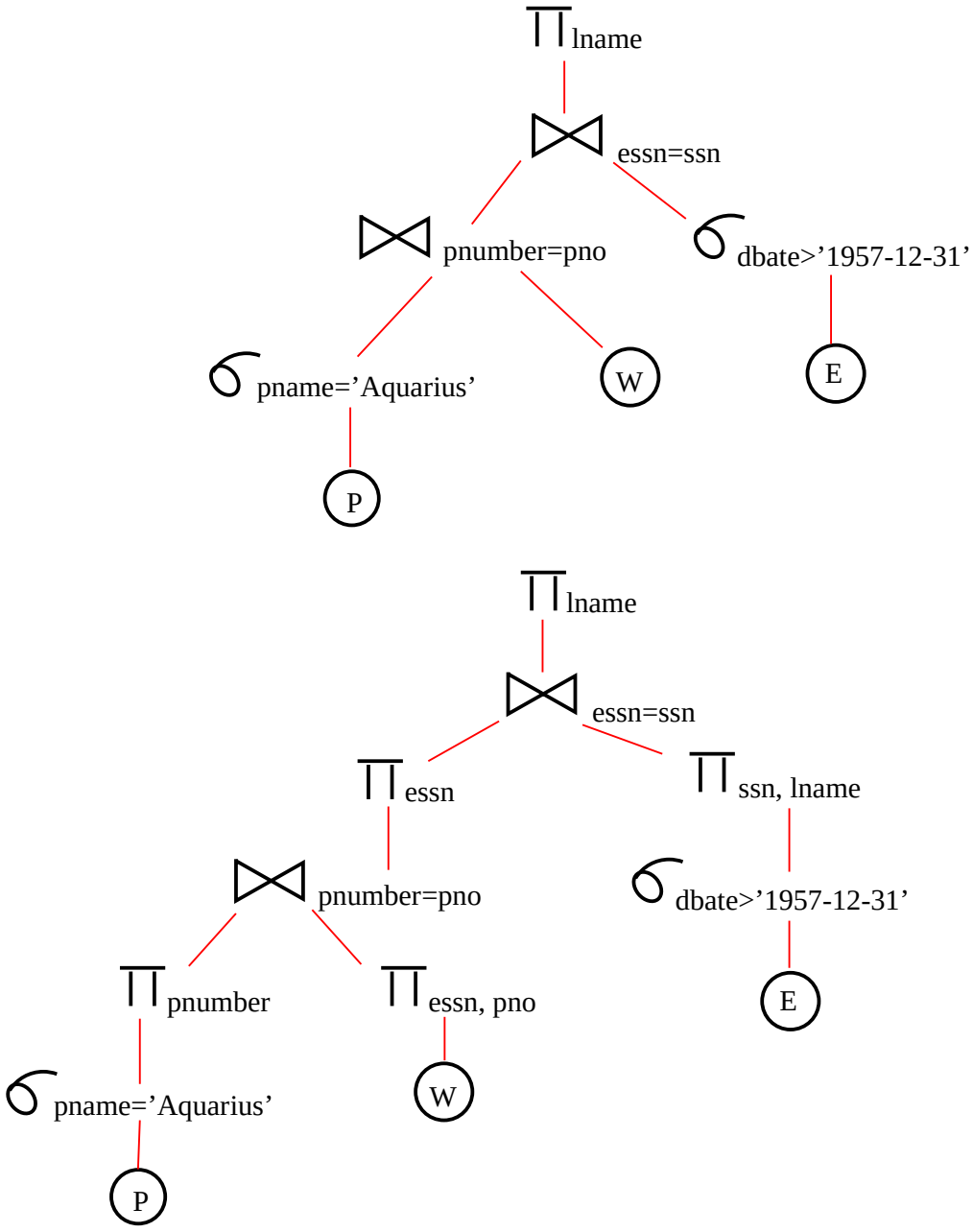


– query tree after leaf nodes re-ordering $\Rightarrow$ query tree after combining $\times, \sigma$ to $\bowtie$.





– query tree after combining $\times, \sigma$ to $\bowtie \Rightarrow$ query tree after pushing down projection operation.



15.8 Using Selectivity and Cost Estimates in Query Optimization

- A query optimizer should not depend solely on heuristic rules; it should also estimate and compare the costs of executing a query using different execution strategies and choose the lowest cost estimate.
- We need a cost function which estimates the costs of executing a query.

15.8.1 Cost Components for Query Execution

- The cost of executing a query includes the following components:
  - **1. Access cost to secondary storage:** The cost of searching for, reading, and writing data blocks that reside on secondary storage.
  - **2. Storage cost:** The cost of storing temporary files generated by an execution strategy for the query.
  - **3. Computation cost:** The cost of performing in-memory operations on the data buffers during query execution.
  - **4. Memory usage cost:** The cost of pertaining to the number of memory buffers needed during query execution.
  - **5. Communication cost:** The cost of shipping the query and its results from the database site to the site or terminal where the query originated.
- Different applications emphasize differently on individual cost components. For example,
  - For large databases, the main emphasis is on minimizing the access cost to secondary storage.
  - For smaller databases, the emphasis is on minimizing computation cost because most of the data in files involved in the query can be completely stored in memory.

- For distributed databases, communication cost must be minimized despite of other factors.
- It is difficult to include all the cost components in a weighted cost function because of the difficulty of assigning suitable weights to the cost components.

15.8.2 Catelog Information Used in Cost Functions

- The necessary information for cost function evaluation is stored in DBMS catelog.
  - The size of each file.
  - For each file, the **number of records (tuples)(r)**, the (average) **record size (R)**, the **number of blocks (b)**, and possibly the **blocking factor (bfr)**.
  - The file records may be unordered, ordered by an attribute with or without a primary or clustering index.
  - For each file, the *access methods or indexes* and the corresponding *access attributes*.
  - The **number of levels (x)** of each multilevel index is needed for cost functions that estimate the number of block accesses.
  - The **number of first-level index blocks ($\mathbf{b}_I1$)**.
  - The **number of distinct values (d)** of an attribute and its **selectivity (sl)**, which is the fraction of records satisfying an equality condition on the attribute.
    - The selectivity allows us to estimate the **selection cardinality ($\mathbf{s = sl \times r}$)** of an attribute, which is the average number of records that will satisfy an equality selectionon that attribute.
    - For a key attribute, **$\mathbf{d = r}$, $\mathbf{sl = 1/r}$ and $\mathbf{s = 1}$**
    - For a nonkey attribute, by making an assumption that the **d** distinct values are uniformly distributed among the records, then **$\mathbf{sl = 1/d}$ and $\mathbf{s = r/d}$**