21CS53: DATABASE MANAGEMENT SYSTEMS

(Effective from the academic year 2021 -2022)

Semester: 05 **Credits: 03**
Number of Contact Hours/Week 3:2:0 **CIE Marks: 50**
Total Number of Hours: 40 **Exam Marks: 50**

MODULE – II

Relational Model

Relational Model Concepts,
Relational Model Constraints and relational database schemas,
Update operations, transactions, and dealing with constraint violations.

Relational Algebra

Unary and Binary relational operations,
Additional relational operations (aggregate, grouping, etc.)
Examples of Queries in relational algebra.

Mapping Conceptual Design into a Logical Design

Relational Database Design using ER-to-Relational mapping.

Chapter 5: The Relational Data Model and Relational Database Constraints

- The relational model was introduced by Tedd Codd of IBM research in 1970 in a classic paper (Codd 1970), and attracted immediate attention due to its simplicity and mathematical foundation
- The model uses the concept of a *mathematical relation*—which looks like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic.
- The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system.
- Current popular relational DBMSs (RDBMSs) include DB2, Oracle etc.
- Data models that preceded the relational model include the hierarchical and network models. These models and systems are now referred to as *legacy database systems.*

5.1 RELATIONAL MODEL CONCEPTS

- **The relational model represents the database as a collection of relations.**
- Informally, each relation resembles a **table of values** or, to some extent, a ***flat* file of records**.
- It is called a **flat file** because each record has a simple linear or *flat* structure however; there are important differences between relations and files.
- When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values.
- A row represents a fact that typically corresponds to a real-world entity or relationship.
- The table name and column names are used to help to interpret the meaning of the values in each row. For Example, in a Student table the column names—Name, Student_number, Class, and Major—specify how to interpret the data values in each row, based on the column value.
- In the formal relational model terminology, a **row is called a tuple**, a **column header is called** an **attribute**, and **the table is called a relation**.
- The data type describing the types of values that can appear in each column is represented by a **domain** of possible values.

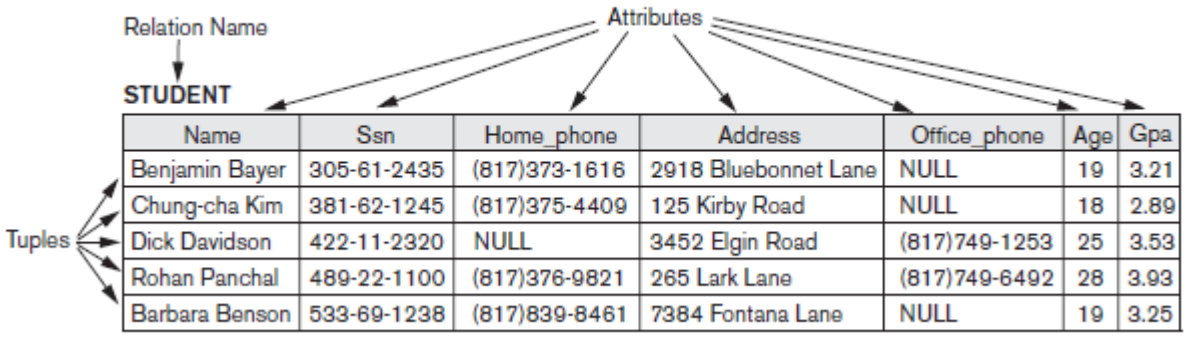
5.1.1Domains, Attributes, Tuples and Relations

- A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned.
- A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn.
- It is also useful to name the domain, to help in interpreting its values. Some examples of domains follow:
  - Usa_phone_numbers. The set of ten-digit phone numbers valid in the United States.
  - Local_phone_numbers. The set of seven-digit phone numbers valid within a particular area code in the United States.
  - Social_security_numbers. The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
  - Names: The set of character strings that represent names of persons.

* Grade_point_averages. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.
* Employee_ages. Possible ages of employees in a company; each must be an integer value between 15 and 80.
* Academic_department_names. The set of academic department names in a university, such as Computer Science, Economics, and Physics.
* Academic_department_codes. The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'.

- The preceding is called *logical* definitions of domains.
- A **data type** or **format** is also specified for each domain.
- For example, the data type for the domain Usa_phone_numbers can be declared as a character string of the form (*ddd*)*ddddddd*, where each *d* is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80.
- **A domain is thus given a name, data type, and format**. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as Person_weights should have the units of measurement, such as pounds or kilograms.
- A **relation schema** R, denoted by $R(A_1, A_2, ...,A_n)$, is made up of a relation name R and a list of attributes, $A_1, A_2, ...,A_n$. Each **attribute** A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by **dom**(A_i).
- The **degree** (or **arity**) of a relation **is the number of attributes *n* of its relation schema**. A relation of degree seven, which stores information about university students, would contain seven attributes describing each student. as follows:
    - STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)
- Using the data type of each attribute, the definition is sometimes written as:
    - STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)
- More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation: dom(Name) = Names; dom(Ssn) = Social_security_numbers; dom(HomePhone) = USA_phone_numbers, dom(office_phone)= USA_phone_numbers, and dom(Gpa) = Grade_point_averages.
- It is also possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is Ssn, whereas the fourth attribute is Address.
- A **relation** (or **relation state**) r of the relation schema $R(A_1, A_2, ...,A_n)$, also denoted by $r(R)$, is a set of n-tuples $r = \{t_1, t_2, ..., t_m\}$. Each ***n*-tuple** t is an ordered list of n values $t = <v_1, v_2, ..., v_n>$, where each value v_i, $1 \leq i \leq n$, is an element of dom (A_i) or is a special NULL value.
- The ith value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.
- Figure 5.1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column.

- *NULL values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1 The attributes and tuples of a relation STUDENT

- The definition of a relation using set theory concepts is as follows: A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains $\text{dom}(A_1)$, $\text{dom}(A_2)$, ..., $\text{dom}(A_n)$, which is a **subset** of the **Cartesian product** (denoted by ×) of the domains that define R:

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \ldots \times \text{dom}(A_n))$$

- The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality,** in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is $|\text{dom}(A_1)| \times |\text{dom}(A_2)| \times \ldots \times |\text{dom}(A_n)|$
- This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. Of all these possible combinations, a relation state at a given time—the **current relation state—reflects only the valid tuples that represent a particular state of the real world**. The schema R is relatively static and changes infrequently.
- It is possible for several attributes to *have* the same domain. The attribute names indicate different **roles**, or interpretations, for the domain. For example, in the STUDENT relation, the same domain USA_phone_numbers plays the role of Home_phone, referring to the *home phone of a student*, and the role of Office_phone, referring to the *office phone of the student*.

5.1.2 Characteristics of Relations

- Certain characteristics make a relation different from a file or table.

1) Ordering of Tuples in a Relation

- A relation is defined as a *set* of tuples.
- Mathematically, **elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order.**
- However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, i^{th}, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.
- Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. For example, tuples in the STUDENT relation in Figure 5.1 could be ordered by values of Name, Ssn, Age, or some other attribute.

- There is *no preference* for one ordering over another. Hence, the relation displayed in Figure 5.2 is considered *identical* to the one shown in Figure 5.1.
- When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

Figure 5.2 The relation STUDENT from figure 5.1 with a different order of tuples

2) Ordering of Values within a Tuple and an Alternative Definition of a Relation

- According to the preceding definition of a relation, an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important.
- However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.
- An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary.* In this definition, a relation schema $R = \{A_1, A_2, ..., A_n\}$ is a *set* of attributes, and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, ..., t_m\}$, where each tuple t_i is a **mapping** from R to D, and D is the **union** of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup ... \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r. Each mapping t_i is called a tuple.
- According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is *not* important, because the *attribute name* appears with its *value*.
- By this definition, the two tuples shown in Figure 5.3 are identical. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appears before another in a tuple.
- When a relation is implemented as a file, the attributes are physically ordered as fields within a record. We will generally use the **first definition** of relation, where the attributes and the values within tuples *are ordered,* because it simplifies much of the notation.

$t = <$ (Name, Dick Davidson),(Ssn, 422-11-2320),(Home_phone, NULL),(Address, 3452 Elgin Road),
(Office_phone, (817)749-1253),(Age, 25),(Gpa, 3.53)>

$t = <$ (Address, 3452 Elgin Road),(Name, Dick Davidson),(Ssn, 422-11-2320),(Age, 25),
(Office_phone, (817)749-1253),(Gpa, 3.53),(Home_phone, NULL)>

Figure 5.3 Two identical tuples when the order of attributes and values is not part of relation definition

3) Values and NULLs in the Tuples

- Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model**.
- Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.
- An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases.
- For example, in Figure 5.1, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*).
- In general, we can have several meanings for NULL values, such as ***value unknown***, ***value*** exists but is ***not available***, or ***attribute does not apply*** to this tuple *(*also known as ***value undefined****).*
- An example of the last type of NULL will occur if we add an attribute Visa_status to the STUDENT relation that applies only to tuples that represents foreign students.

4) Interpretation of meaning of a relation

- The relation schema can be interpreted as a declaration or a type of **assertion**.
- For example, the schema of the STUDENT relation of Figure 5.1 asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion.
- Notice that some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.* For example, a relation schema MAJORS(Student_ssn, Department_code) asserts that students major in academic disciplines. A tuple in this relation relates a student to his or her major discipline.
- This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type.
- An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. For example, the predicate STUDENT (Name, Ssn, ...) is true for the five tuples in relation STUDENT of Figure 5.1. These tuples represent five different propositions or facts in the real world. This interpretation is quite useful in the context of logical programming languages, such as Prolog, because it allows the relational model to be used within these languages.
- An assumption called **the closed world assumption** states that the only true facts in the universe are those present within the extension (state) of the relation(s). Any other combination of values makes the predicate false.

5.1.3 Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, ..., A_n)$.
- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.

- In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation—the *current relation state*—whereas STUDENT (Name, Ssn, ...) refers *only* to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation $R.A$—for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.
- An n-tuple t in a relation $r(R)$ is denoted by $t = <v_1, v_2, ..., v_n>$, where v_i is the value corresponding to attribute A_i. The following notation refers to **component values** of tuples:
    - Both $t[A_i]$ and $t.A_i$ (and sometimes $t[i]$) refer to the value v_i in t for attribute A_i.
    - Both $t[A_u, A_w, ..., A_z]$ and $t.(A_u, A_w, ..., A_z)$, where $A_u, A_w, ..., A_z$ is a list of attributes from R, refer to the subtuple of values $<v_u, v_w, ..., v_z>$ from t corresponding to the attributes specified in the list.

5.2 RELATIONAL MODEL CONSTRAINTS AND RELATIONAL DATABASE SCHEMAS

- The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents.
- In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints.
- Constraints on databases can generally be divided into three main categories:
    1. Constraints that is inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
    2. Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL (data definition language, see Section2.3.1).We call these **schema-based constraints** or **explicit constraints**.
    3. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. These **application-based** or **semantic constraints** or **business rules**.
- Another important category of constraints is *data dependencies*, which include *functional dependencies* and *multivalued dependencies*. They are used mainly for testing the "goodness" of the design of a relational database and are utilized in a process called *normalization.*
- We discuss the main types of constraints that can be expressed in the relational model the schema-based constraints from the second category include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

5.2.1 Domain Constraints

- Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain dom(A).
- The data types associated with domains typically include **standard** numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double precision float).
- Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types.

- Other possible domains may be described by a **subrange** of values from a data type or as an **enumerated** data type in which all possible values are explicitly listed.

5.2.2 Key Constraints and Constraints on NULL values

- A relation is defined as a set of tuples. All elements of a set are distinct; hence, all tuples in a relation must also be distinct.
- This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes.

Super Key:

- Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples $t1$ and $t2$ in a relation state r of R, we have the constraint that:

$$t1[SK] \neq t2[SK]$$

- Any such set of attributes SK is called a **superkey** of the relation schema R.
- A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state r of R can have the same value for SK.
- Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes.

Key:

- A *key* has no redundancy. A **key** K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more.
- Hence, **a key satisfies two properties:**
    1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to superkey.
    2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold.
- Whereas the first property applies to both keys and superkeys, the second property is required only for keys. Hence, a key is also a superkey but not vice versa.
- Consider the STUDENT relation of Figure 5.1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn. Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey.
- In general, any superkey formed from a single attribute is also a key.
- A key with multiple attributes must require *all* its attributes together to have the uniqueness property.
- The value of a key attribute can be used to identify uniquely each tuple in the relation.
- For example, the Ssn value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation.

- Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema.
- A key is determined from the meaning of the attributes, and the property is *time-invariant:* It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 5.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.

Candidate Key & Unique key:

- A **relation schema may have more than one key. In this case, each of the keys is called a candidate key.**
- For example, the CAR relation in Figure 5.4 has two candidate keys: License_number and Engine_serial_number.
- It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 5.4.
- Notice that when a relation schema has several candidate keys, the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys**, and are not underlined.

Constraints on NULL values:

- Another constraint on attributes specifies whether NULL values are or are not permitted.
- For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4 The CAR relation, with two candidate keys: License_number and Engine_serial_number

5.2.3 Relational Databases and Relational Database Schemas

- The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema.
- A **relational database schema** S is a set of relation schemas S = $\{R_1, R_2, ..., R_m\}$ and a set of **integrity constraints** IC.
- A **relational database state** DB of S is a set of relation states DB = $\{r_1, r_2, ..., r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC.
- Figure 5.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. The underlined attributes represent primary keys.
- Figure 5.6 shows a relational database state corresponding to the COMPANY schema.
- When we refer to a relational database, we implicitly include both its schema and its current state.

- A database state that does not obey all the integrity constraint is called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**
- In Figure 5.5, the Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. **Attributes that represent the same real-world concept may or may not have identical names in different relations.**
- Alternatively**, attributes that represent different concepts may have the same name in different relations.** For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different realworld concepts—project names and department names.
- In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation of Figure 5.5: once in the role of the employee's SSN, and once in the role of the supervisor's SSN.
- We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.
- Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose.
- Integrity constraints are specified on a database schema and are expected to hold on every valid database state of that schema. In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

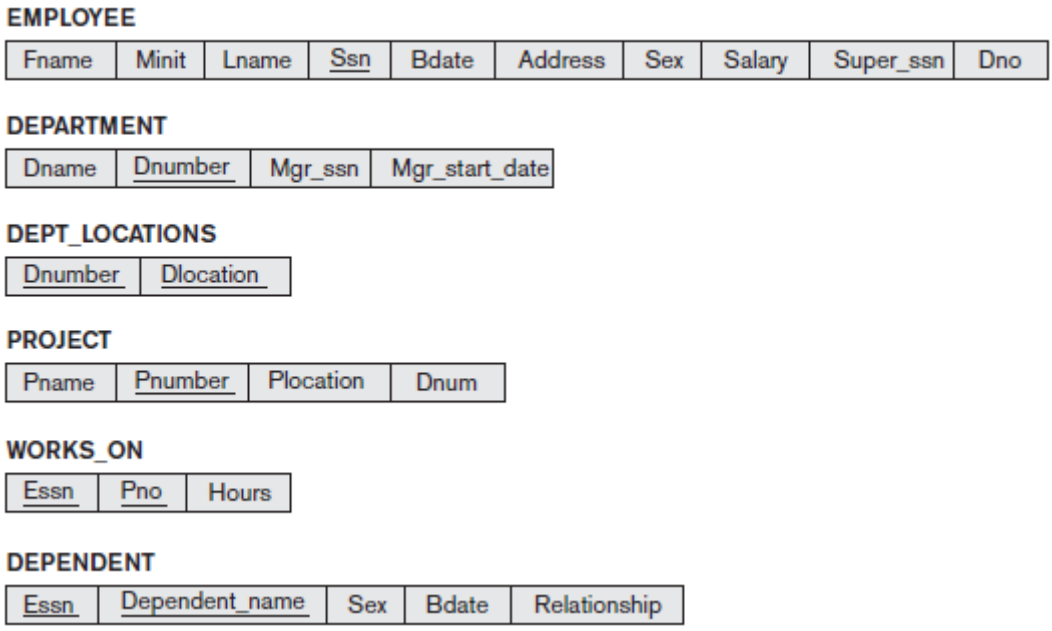


Figure 5.5 Schema Diagram for the Company Relational Database Schema

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

- The **entity integrity constraint** states that **no primary key value can be NULL**. This is because the primary key value is used to identify individual tuples in a relation.
- Having NULL values for the primary key implies that we cannot identify some tuples.
- For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

- Key constraints and entity integrity constraints are specified on individual relations.
- The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations.
- Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation.
- For example, in Figure 5.6, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.
- To define referential integrity more formally, first we define the concept of a *foreign key.*

Foreign Key:

- The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:
  1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.
  2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ *or is NULL*.
- In the former case, we have t_1[FK] = t_2[PK], and we say that the tuple t_1 **references** or **refers to** the tuple t_2.
- In this definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation**.
- If these two conditions hold, a **referential integrity constraint** from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.
- To specify these constraints, first we must have a clear understanding of the meaning or role that each attribute or set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas.
- For example, consider the database shown in Figure 5.6. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of Dno *can be NULL* if the employee does not belong to a department or will be assigned to a department later.
- For example, in Figure 5.6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department.
- Notice that a foreign key can *refer to its own relation.*
- For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure 5.6 the tuple for employee 'John Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith.'
- **We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation**. Figure 5.7 shows the schema in Figure 5.5 with the referential integrity constraints displayed in this manner.

- All integrity constraints should be specified on the relational database schema (i.e., defined as part of its definition) if we want to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them.
- Most relational DBMSs support key, entity integrity, and referential integrity constraints. These constraints are specified as a part of data definition in the DDL.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Figure 5.6 One possible database states for the company relational database schema

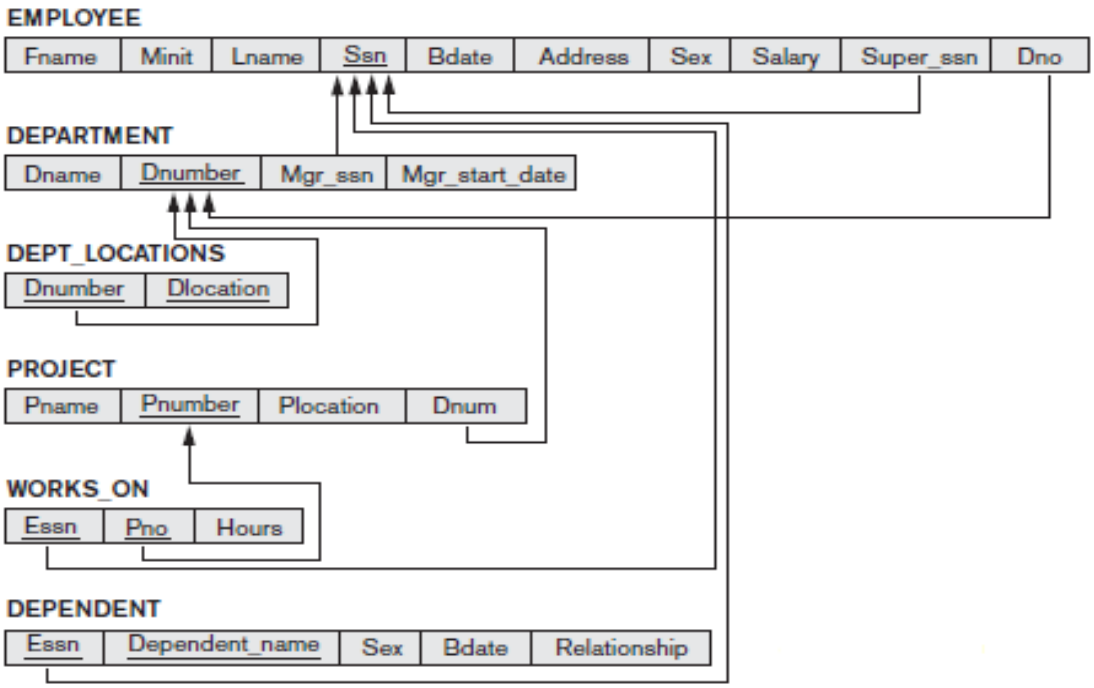


Figure 5.7 Referential Integrity Constraints displayed on the COMPANY Relational Database Schema

5.2.5 Other Types of Constraints

- The preceding integrity constraints are included in the data definition language because they occur in most database applications.
- However, they do not include a large class of general constraints, sometimes called ***semantic integrity constraints***, which may have to be specified and enforced on a relational database. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee's supervisor* and *the maximum number of hours an employee can work on all projects per week is 56*.
- Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**.
- Mechanisms called **triggers** and **assertions** can be used. In SQL, CREATE ASSERTION and CREATE TRIGGER statements can be used for this purpose.
- It is more common to check for these types of constraints within the application programs than to use constraint specification languages because the latter are sometimes difficult and complex to use
- Another type of constraint is the ***functional dependency*** **constraint**, which establishes a functional relationship among two sets of attributes X and Y.
- This constraint specifies that the value of X determines a unique value of Y in all states of a relation; it is denoted as a functional dependency $X \rightarrow Y$. We use functional dependencies and other types of dependencies in Chapters 15 and 16 as tools to analyze the quality of relational designs and to "normalize" relations to improve their quality.
- The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy.
- Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.

- An example of a transition constraint is: “the salary of an employee can only increase.” Such constraints are typically enforced by the application programs or specified using active rules and triggers.

5.3 Update Operations, Transactions, and Dealing with Constraint Violations

- There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records.
- Insert is used to insert one or more new tuples in a relation.
- Delete is used to delete tuples.
- Update (or Modify) is used to change the values of some attributes in existing tuples.
- Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated.
- We use the database shown in Figure 5.6 for examples and discuss only key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 5.7

5.3.1 The Insert Operation

- The Insert operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R. Insert can violate any of the four types of constraints discussed in the previous section.
    - Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type.
    - Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R).
    - Entity integrity can be violated if any part of the primary key of the new tuple t is NULL.
    - Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation.
- Here are some examples to illustrate this discussion.
    - Operation:
        - Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, NULL, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4> into EMPLOYEE.
        - Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.
    - Operation:
        - Insert <‘Alicia’, ‘J’, ‘Zelaya’, ‘999887777’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, ‘987654321’, 4> into EMPLOYEE.
        - Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.
    - Operation:
        - Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windswept, Katy, TX’, F, 28000, ‘987654321’, 7> into EMPLOYEE.
        - Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.
    - Operation:

- Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4> into EMPLOYEE.
- Result: This insertion satisfies all constraints, so it is acceptable.

- If an insertion violates one or more constraints, the default option is to reject the insertion.

5.3.2 The Delete Operation

- The Delete operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database.
- To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.
  - Operation:
    - Delete the WORKS_ON tuple with Essn = ‘999887777’ and Pno = 10.
    - Result: This deletion is acceptable and deletes exactly one tuple.
  - Operation:
    - Delete the EMPLOYEE tuple with Ssn = ‘999887777’.
    - Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.
  - Operation:
    - Delete the EMPLOYEE tuple with Ssn = ‘333445555’.
    - Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT,WORKS_ON, and DEPENDENT relations.
- Several options are available if a deletion operation causes a violation.
  - The first option, called restrict, is to reject the deletion.
  - The second option, called cascade, is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = ‘999887777’.
  - A third option, called set null or set default, is to modify the referencing attribute values that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple. Notice that if a referencing attribute that causes a violation is part of the primary key, it cannot be set to NULL; otherwise, it would violate entity integrity.
- Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = ‘333445555’. Tuples in EMPLOYEE with Super_ssn = ‘333445555’ and the tuple in DEPARTMENT with Mgr_ssn = ‘333445555’ can have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.
- In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to specify which of the options applies in case of a violation of the constraint.

.

5.3.3 The Update Operation

- The Update (or Modify) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation R. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified.
- Here are some examples.
    - Operation:
        - Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
        - Result: Acceptable.
    - Operation:
        - Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
        - Result: Acceptable.
    - Operation:
        - Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.
        - Result: Unacceptable, because it violates referential integrity.
    - Operation:
        - Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.
        - Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.
- Updating an attribute that is neither part of a primary key nor of a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain.
- Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples.
- If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL).
- Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation.

5.3.4 The Transaction Concept

- A database application program running against a relational database typically executes one or more transactions.
- A transaction is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database.
- At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema.
- A single transaction may involve any number of retrieval operations and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database.
- For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.
- A large number of commercial applications running against relational databases in online transaction processing (OLTP) systems are executing transactions at rates that reach several hundred per second.

The Relational Algebra

- In this chapter, we discuss the two *formal languages* for the relational model: the relational algebra and the relational calculus.
- A data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining the database's structure and constraints.
- The basic set of operations for the relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as *relational algebra expressions*. The result of retrieval is a new relation, which may have been formed from one or more relations.
- A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

- The relational algebra is very important for several reasons:
  - First, it provides a formal foundation for relational model operations.
  - Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs).
  - Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs.

- The **relational algebra** is often considered to be an integral part of the relational data model. Its operations can be **divided into two groups**.
  - One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the *formal* relational model. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT (also known as CROSS PRODUCT).
  - The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. First, we describe the SELECT and PROJECT operations.
- The **algebra** defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries.
- A relational calculus expression creates a new relation. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus.
- The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in a variation of relational calculus known as the tuple relational calculus.

6.1 Unary Relational Operations: SELECT and PROJECT

6.1.1 The SELECT Operation

- The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**.
- One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition.
- Alternatively, we can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition.
- The SELECT operation can also be visualized as a ***horizontal partition* of the relation** into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition are discarded.
- For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$$\sigma_{Dno=4}(EMPLOYEE)$$
$$\sigma_{Salary>30000}(EMPLOYEE)$$

- In general, the SELECT operation is denoted by

$$\sigma_{<\text{selection condition}>}(R)$$

where the symbol **σ (sigma)** is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*.

- Notice that *R* is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation.
- The relation resulting from the SELECT operation has the *same attributes* as *R*.
- The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

  <attribute name> <comparison op> <constant value>

  or

  <attribute name> <comparison op> <attribute name>

  where <attribute name> is the name of an attribute of *R*, <comparison op> is normally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant value from the attribute domain. Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition.
- For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

  $$\sigma_{(Dno=4\ \textbf{AND}\ Salary>25000)\ \textbf{OR}\ (Dno=5\ \textbf{AND}\ Salary>30000)}(EMPLOYEE)$$

  The result is shown in Figure 6.1(a).
- Notice that all the comparison operators in the set {=, <, ≤, >, ≥, ≠} can apply to attributes whose domains are *ordered values*, such as numeric or date domains.

- Domains of strings of characters are also considered to be ordered based on the collating sequence of the characters.
- If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set {=, ≠} can be used. An example of an unordered domain is the domain Color = { 'red', 'blue', 'green', 'white', 'yellow',...}, where no order is specified among the various colors.
- Some domains allow additional types of comparison operators; for example, a domain of character strings may allow the comparison operator SUBSTRING_OF.
- In general, the result of a SELECT operation can be determined as follows.
- The <selection condition> is applied independently to each *individual tuple t* in *R*. This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to TRUE, then tuple *t* is **selected**. All the selected tuples appear in the result of the SELECT operation.
- The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:
    1. (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
    2. (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
    3. (**NOT** cond) is TRUE if cond is FALSE; otherwise, it is FALSE.
- The SELECT operator is **unary**; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple.
- The **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of *R*. The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in *R*. That is, $|\sigma_c (R)| \leq |R|$ for any condition *C*.
- The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.
- Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

- Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ \ldots)) = \sigma_{<cond1>\ \textbf{AND}<cond2>\ \textbf{AND}\ldots\textbf{AND}\ <cond\mathit{n}>}(R)$$

6.1.2 The Project Operation

- The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns.
- The result of the PROJECT operation can be visualized as a ***vertical partition*** of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns.
- For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$$

- The resulting relation is shown in Figure 6.1(b). The general form of the PROJECT operation is

$$\pi_{\text{<attribute list>}}(R)$$

- where **π (pi)** is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation *R*. Notice that *R* is, in general, a *relational algebra expression* whose result is a relation.

- The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.

- If the attribute list includes only nonkey attributes of *R*, duplicate tuples are likely to occur. The **PROJECT operation *removes any duplicate tuples***, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**.

- For example, consider the following PROJECT operation:

$$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

- The result is shown in Figure 6.1(c). Notice that the tuple <'F', 25000> appears only once in Figure 6.1(c), even though this combination of values appears twice in the EMPLOYEE relation. Duplicate elimination involves sorting or some other technique to detect duplicates and thus adds more processing.

- If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. This was not permitted in the formal relational model, but is allowed in SQL.

- The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in *R*.

- If the projection list is a superkey of *R*—that is, it includes some key of *R*—the resulting relation has the *same number* of tuples as *R*.

- Moreover, $\pi_{\text{<list1>}}(\pi_{\text{<list2>}}(R)) = \pi_{\text{<list1>}}(R)$ as long as <list2> contains the attributes in <list1>;otherwise, the left-hand side is an incorrect expression.

- It is also noteworthy that commutativity *does not* hold on PROJECT.

Figure 6.1
Results of SELECT and PROJECT operations. (a) $\sigma_{(Dno=4\ \textbf{AND}\ Salary>25000)\ OR\ (Dno=5\ \textbf{AND}\ Salary>30000)}$ (EMPLOYEE). (b) $\pi_{Lname,\ Fname,\ Salary}$(EMPLOYEE). (c) $\pi_{Sex,\ Salary}$(EMPLOYEE).

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

6.1.3 Sequences of Operations and the RENAME operation

- In general, for most queries, we need to apply several relational algebra operations one after the other. 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.
- In the latter case, we must give names to the relations that hold the intermediate results. For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation.
- We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

$$\pi_{Fname,\ Lname,\ Salary}(\sigma_{Dno=5}(EMPLOYEE))$$

- Figure 6.2(a) shows the result of this in-line relational algebra expression.
- Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Fname,\ Lname,\ Salary}(DEP5\_EMPS)$$

- It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.
- We can also use this technique to **rename** the attributes in the intermediate and result relations.
- To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$TEMP \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$R_{(First\_name,\ Last\_name,\ Salary)} \leftarrow \pi_{Fname,\ Lname,\ Salary}(TEMP)$$

- These two operations are illustrated in Figure 6.2(b).
- If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order.
- For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

- We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator.
- The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

  $\rho_{S(B1, B2, ..., Bn)}(R)$ or $\rho_{S}(R)$ or $\rho_{(B1, B2, ..., Bn)}(R)$

  where the symbol **ρ (rho)** is used to denote the RENAME operator, S is the new relation name, and $B1, B2, ..., Bn$ are the new attribute names.
- The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are $(A1, A2, ..., An)$ in that order, then each Ai is renamed as Bi.

(a)

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston,TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble,TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figure 6.2
Results of a sequence of operations. (a) $\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$. (b) Using intermediate relations and renaming of attributes.

6.2 Relational Algebra Operations from Set Theory

6.2.1 The UNION, INTERSECTION, and MINUS Operations

- The next group of relational algebra operations are the standard mathematical operations on sets. For example, to retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

  $DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$

  $RESULT1 \leftarrow \pi_{Ssn}(DEP5\_EMPS)$

  $RESULT2(Ssn) \leftarrow \pi_{Super\_ssn}(DEP5\_EMPS)$

  $RESULT \leftarrow RESULT1 \cup RESULT2$
- The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 6.3), while eliminating any duplicates. Thus, the Ssn value '333445555' appears only once in the result.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

Figure 6.3
Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2.

- Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**). These are **binary** operations; that is, each is applied to two sets (of tuples).
- When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility* or *type compatibility*.
- Two relations $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_n)$ are said to be **union compatible** (or **type compatible**) if they have the same degree n and if $dom(A_i) = dom(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.
- We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:
    1. UNION: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
    2. INTERSECTION: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
    3. SET DIFFERENCE (or MINUS): The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.
- We will adopt the convention that the resulting relation has the same attribute names as the *first* relation R. It is always possible to rename the attributes in the result using the rename operator.
- Figure 6.4 illustrates the three operations.
- The relations STUDENT and INSTRUCTOR in Figure 6.4(a) are union compatible and their tuples represent the names of students and the names of instructors, respectively.
- The result of the UNION operation in Figure 6.4(b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 6.4(c)) includes only those who are both students and instructors.
- Notice that both UNION and INTERSECTION are *commutative operations*; that is,
    - $R \cup S = S \cup R$ and $R \cap S = S \cap R$
- Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are also *associative operations;* that is,
    - $R \cup (S \cup T) = (R \cup S) \cup T$ and $(R \cap S) \cap T = R \cap (S \cap T)$
- The MINUS operation is *not commutative;* that is, in general, $R - S \neq S - R$
- Figure 6.4(d) shows the names of students who are not instructors, and Figure 6.4(e) shows the names of instructors who are not students.
- Note that INTERSECTION can be expressed in terms of union and set difference as follows: $R \cap S = ((R \cup S) - (R - S)) - (S - R)$

Figure 6.4
The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

6.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) OPERATION

- Cross product is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible.
- In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set).
- In general, the result of $R(A_1, A_2, ..., A_n) \times S(B_1, B_2, ..., B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, in that order.
- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.
- It is mostly useful when followed by a selection that matches values of attributes coming from the component relations.
- For example, suppose that we want to retrieve a list of names of each female employee's dependents.
- We can do this as follows:

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex='F'}}(\text{EMPLOYEE})$$

$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$

$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

- The resulting relations from this sequence of operations are shown in Figure 6.5. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 6.5 with DEPENDENT.

- In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful (every dependent is combined with *every* female employee).
- We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn value match the Ssn value of the EMPLOYEE tuple.
- The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all. It is the responsibility of the user to make sure to apply only meaningful operations to relations.
- The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example.
- Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *related tuples* from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.

Figure 6.5
The Cartesian Product (Cross Product) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

6.3 Binary Relational Operations: Join and Division

6.3.1 The Join Operation

- The **JOIN** operation, denoted by $\bowtie$, is used to combine *related tuples* from two relations into single "longer" tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations.
- To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple.
- We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$$
$$\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$$

- The first operation is illustrated in Figure 6.6. Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation.
- This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.
- The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$

- These two operations can be replaced with a single JOIN operation as follows:

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \text{EMPNAMES} \bowtie_{\text{Ssn=Essn}} \text{DEPENDENT}$$

- The general form of a JOIN operation on two relations $R(A1, A2, ..., An)$ and $S(B1, B2, ..., Bm)$ is

$$R \bowtie_{\text{<join condition>}} S$$

- The result of the JOIN is a relation Q with $n + m$ attributes $Q(A1, A2, ..., An, B1, B2, ... , Bm)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition*. This is the main difference between CARTESIAN PRODUCT and JOIN.
- **In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result.**
- The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q *as a single combined tuple*.
- A general join condition is of the form

<condition> **AND** <condition> **AND**...**AND** <condition>

where each <condition> is of the form $Ai\ \theta\ Bj$, Ai is an attribute of R, Bj is an attribute of S, Ai and Bj have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$.

- A JOIN operation with such a general join condition is called a **THETA JOIN**.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

Figure 6.6
Result of the JOIN operation DEPT_MGR ← DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE.

- Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result. In that sense, the join operation does not necessarily preserve all of the information in the participating relations.

6.3.2 Variations of Join: The EQUIJOIN and NATURAL JOIN

- The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the **only comparison operator used is =,** is called an **EQUIJOIN**. Both previous examples were EQUIJOINs.
- Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple. For example, in Figure 6.6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.**
- The standard definition of NATURAL JOIN **requires** that the **two join attributes** (or each pair of join attributes) have the **same name in both relations**. If this is not the case, a renaming operation is applied first.
- Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

$$PROJ\_DEPT \leftarrow PROJECT * \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$

- The same query can be done in two steps by creating an intermediate table DEPT as follows:

$$DEPT \leftarrow \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$
$$PROJ\_DEPT \leftarrow PROJECT * DEPT$$

- The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 6.7(a).
- In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but *only one join attribute value* is kept.

- If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

  DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS
- The resulting relation is shown in Figure 6.7(b), which combines each department with its locations and has one tuple for each location.
- In general, the join condition for NATURAL JOIN is constructed by equating *each pair of join attributes* that have the same name in the two relations and combining these conditions with **AND**.
- A more general, *but nonstandard* definition for NATURAL JOIN is

  $Q \leftarrow R *_{(<list1>),(<list2>)} S$
- In this case, <list1> specifies a list of i attributes from R, and <list2> specifies a list of i attributes from S. The lists are used to form equality comparison conditions between pairs of corresponding attributes, and the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R—<list1>— is kept in the result Q.
- If no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples.
- In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie_{<\text{join condition}>} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity**, which is a property of each join condition.
- **If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.**
- As we can see, a single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called *outer joins.*
- Informally, **an *inner join* is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION**.
- Note that sometimes a join may be specified between a relation and itself.
- The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

  $((\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$
- This links each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager information.

(a)

PROJ_DEPT

Pname	Pnumber	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 6.7
Results of two NATURAL JOIN operations. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

6.3.3 A Complete Set of Relational Algebra Operations

- It has been shown that the set of relational algebra operations {σ, π, U, –, ×} is a **complete** set; that is, any of the other original relational algebra operations can be expressed as a *sequence of operations from this set*.
- For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

- Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection.
- As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation:

$$R \bowtie_{\text{<condition>}} S = \sigma_{\text{<condition>}}(R \times S)$$

- Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.

6.3.4 The Division Operation

- The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications.
- An example is *Retrieve the names of employees who work on **all** the projects that 'John Smith' works on*.
- To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$\text{SMITH} \leftarrow \sigma_{\text{Fname='John' AND Lname='Smith'}}(\text{EMPLOYEE})$$

$$\text{SMITH\_PNOS} \leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} \bowtie_{\text{Essn=Ssn}} \text{SMITH})$$

- Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$SSN\_PNOS \leftarrow \pi_{Essn,\ Pno}(WORKS\_ON)$$

- Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$SSNS(Ssn) \leftarrow SSN\_PNOS \div SMITH\_PNOS$$

$$RESULT \leftarrow \pi_{Fname,\ Lname}(SSNS * EMPLOYEE)$$

- The preceding operations are shown in Figure 6.8(a).

- In general, **the DIVISION operation is applied to two relations *R*(*Z*) ÷ *S*(*X*), where** the attributes of *R* are a subset of the attributes of *S*; that is, ***X*** $\subseteq$ ***Z*.**
- Let *Y* be the set of attributes of *R* that are not attributes of *S*; that is, $Y = Z - X$ (and hence $Z = X \cup Y$). The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ for *every* tuple t_S in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every tuple* in S.
- Note that in the formulation of the DIVISION operation, the tuples in the denominator relation *S* restrict the numerator relation *R* by selecting those tuples in the result that match all values present in the denominator.
- It is not necessary to know what those values are as they can be computed by another operation, as illustrated in the SMITH_PNOS relation in the above example.
- Figure 6.8(b) illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$.
- Notice that the tuples (values) b_1 and b_4 appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: b_2 does not appear with a_2, and b_3 does not appear with a_1.
- The DIVISION operation can be expressed as a sequence of π, $\times$, and – operations as follows:

$$T1 \leftarrow \pi_Y(R)$$

$$T2 \leftarrow \pi_Y((S \times T1) - R)$$

$$T \leftarrow T1 - T2$$

- Most RDBMS implementations with SQL as the primary query language do not directly implement division. SQL has a roundabout way of dealing with the type of query illustrated above. Table 6.1 lists the various basic relational algebra operations we have discussed.

Figure 6.8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

6.3.5 Notation for Query Trees

- In this section, we describe a notation typically used in relational systems to represent queries internally.
- The notation is called a ***query tree*** or sometimes it is known as a ***query evaluation tree*** or ***query execution tree***. It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.
- **A query tree is a tree data structure that corresponds to a relational algebra expression.**
- It represents the **input relations of the query as *leaf nodes* of the tree**, and represents the **relational algebra operations as internal nodes**.
- An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. **The execution terminates when the root node is executed and produces the result relation for the query**.
- Figure 6.9 shows a query tree for Query 2: *For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.* This query is specified on the relational schema of Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<selection condition>}}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<attribute list>}}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\text{<join condition>}} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<join condition>}} R_2$, OR $R_1 \bowtie_{\text{(<join attributes 1>), (<join attributes 2>)}} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<join condition>}} R_2$, OR $R_1 *_{\text{(<join attributes 1>), (<join attributes 2>)}} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

- In Figure 6.9, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE.
- The relational algebra operations in the expression are represented by internal tree nodes. The query tree signifies an explicit order of execution in the following sense.
- In order to execute Q2, the node marked (1) in Figure 6.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2).
- Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on. In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra.

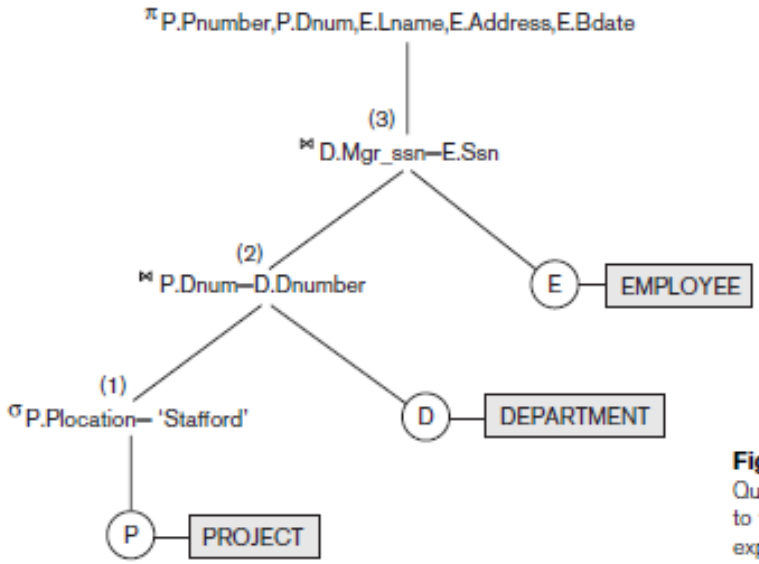


Figure 6.9
Query tree corresponding to the relational algebra expression for Q2.

6.4 Additional Relational Operations

- Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations described in Sections 6.1 through 6.3.
- In this section, we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

6.4.1 Generalized Projection

- The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list. The generalized form can be expressed as:

$$\pi_{F1,\,F2,\,\ldots,\,Fn}(R)$$

where F_1, F_2, ..., F_n are functions over the attributes in relation R and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result.

- As an example, consider the relation
  EMPLOYEE (Ssn, Salary, Deduction, Years_service).
- A report may be required to show

Net Salary = Salary – Deduction
Bonus = 2000 * Years_service, and
Tax = 0.25 * Salary.

- Then a generalized projection combined with renaming may be used as follows:

$$\text{REPORT} \leftarrow \rho_{(\text{Ssn,Net\_salary,Bonus,Tax})}(\pi_{\text{Ssn, Salary–Deduction ,2000*Years\_service,0.25*Salary}}(\text{EMPLOYEE}))$$

6.4.2 Aggregate Functions and Grouping

- Another type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database.
- Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples.
- These functions are used in simple statistical queries that summarize information from the database tuples.
- Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.
- Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function *independently to each group*.
- An example would be to group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department.
- We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.
- We can define an AGGREGATE FUNCTION operation, using the symbol $\mathfrak{I}$ (pronounced *script F*), to specify these types of requests as follows:

$$\text{<grouping attributes>}\ \mathfrak{I}\ \text{<function list>}\ (R)$$

- where <grouping attributes> is a list of attributes of the relation specified in *R*, and <function list> is a list of (<function> <attribute>) pairs.
- In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and <attribute> is an attribute of the relation specified by *R*.
- The resulting relation has the grouping attributes plus one attribute for each element in the function list.
- For example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write: $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}(\text{Dno}\ \mathfrak{I}\ \text{COUNT Ssn, AVERAGE Salary}\ (\text{EMPLOYEE}))$
- The result of this operation on the EMPLOYEE relation of Figure 5.6 is shown in Figure 6.10(a).
- In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation *R*. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>.
- \For example, Figure 6.10(b) shows the result of the following operation:

$$\text{Dno}\ \mathfrak{I}\ \text{COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE})$$

- If no grouping attributes are specified, the functions are applied to *all the tuples* in the relation, so the resulting relation has a *single tuple only*.

- For example, Figure 6.10(c) shows the result of the following operation:

$$\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$$

- It is important to note that, in general, **duplicates are *not eliminated* when an aggregate function is applied;** this way, the normal interpretation of functions such as SUM and AVERAGE is computed**. The result of applying an aggregate function is a relation, not a scalar number—even if it has a single value. This makes the relational algebra a closed system**.

Figure 6.10
The aggregate function operation.
a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}({}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE}))$.
b. ${}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.
c. $\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

6.4.3 Recursive Closure Operations

- Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure.** This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor.
- This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor).
- An example of a recursive operation is to retrieve all supervisees of an employee *e* at all levels—that is, all employees *e*' directly supervised by *e*, all employees *e*'' directly supervised by each employee *e*' all employees *e*''' directly supervised by each employee *e*'', and so on.
- It is relatively straightforward in the relational algebra to specify all employees supervised by *e at a specific level* by joining the table with itself one or more times.
- However, it is difficult to specify all supervisees at *all* levels. For example, to specify the Ssns of all employees *e*' directly supervised—*at level one*—by the employee *e* whose name is 'James Borg' (see Figure 5.6), we can apply the following operation:

$$\text{BORG\_SSN} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname='James' AND name='Borg'}}(\text{EMPLOYEE}))$$

$$\text{SUPERVISION(Ssn1, Ssn2)} \leftarrow \pi_{\text{Ssn,Super\_ssn}}(\text{EMPLOYEE})$$

$$\text{RESULT1(Ssn)} \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{BORG\_SSN})$$

- To retrieve all employees supervised by Borg at level 2—that is, all employees *e*''supervised by some employee *e*' who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

$$\text{RESULT2(Ssn)} \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{RESULT1})$$

- To get both sets of employees supervised at levels 1 and 2 by 'James Borg', we can apply the UNION operation to the two results, as follows:

  RESULT ← RESULT2 U RESULT1
- The results of these queries are illustrated in Figure 6.11. Although it is possible to retrieve employees at each level and then take their UNION, we cannot, in general, specify a query such as "retrieve the supervisees of 'James Borg' at all levels" without utilizing a looping mechanism unless we know the maximum number of levels.
- An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

SUPERVISION

(Borg's Ssn is 888665555)

(Ssn)	(Super_ssn)
Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by Borg's subordinates)

RESULT

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 ∪ RESULT2)

Figure 6.11
A two-level recursive query.

6.4.4 OUTER JOIN Operations

- Next, we discuss some additional extensions to the JOIN operation that are necessary to specify certain types of queries.
- The JOIN operations described earlier match tuples that satisfy the join condition.
- For example, for a NATURAL JOIN operation $R * S$, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result.
- Tuples with NULL values in the join attributes are also eliminated. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.
- A set of operations, called **outer joins**, were developed for the case **where the user wants to keep all the tuples in *R*, or all those in *S*, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation**. This satisfies the

need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values. **The join operations** described in section 6.3, **where only matching tuples are kept in the result, are called inner joins**.

- For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*; if they do not manage one, we can indicate it with a NULL value.
- We can apply an operation **LEFT OUTER JOIN**, denoted by , ⟕ to retrieve the result as follows:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} \text{ ⟕ }_{\text{Ssn=Mgr\_ssn}} \text{DEPARTMENT})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$$

The LEFT OUTER JOIN operation keeps every tuple in the *first*, or *left*, relation R in R ⟕ S; if no matching tuple is found in S, then the attributes of S in the join result are filled or *padded* with NULL values. The result of these operations is shown in Figure 6.12.

- A similar operation, **RIGHT OUTER JOIN**, denoted by , ⟖ keeps every tuple in the second, or right, relation S in the result ofR ⟖ S.
- A third operation, **FULL OUTER JOIN**, denoted by , ⟗, keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

Figure 6.12
The result of a LEFT OUTER JOIN operation.

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

6.4.5 The OUTER UNION Operations

- The **OUTER UNION** operation was developed to take the union of tuples from two relations that have some common attributes, but are *not union (type) compatible*.
- This operation will take the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible**, meaning that only some of their attributes, say X, are union compatible. The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation $T(X, Y, Z)$. It is therefore the same as a FULL OUTER JOIN on the common attributes.
- Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X]=t_2[X]$. These will be combined (unioned) into a single tuple in t. Tuples in either relation that have no matching tuple in the other relation are padded with NULL values.
- For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, Ssn, Department, Advisor) and INSTRUCTOR(Name, Ssn, Department, Rank). Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department. The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

  STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)

- All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute. A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes.
- Notice that the same person may still appear twice in the result. For example, we could have a graduate student in the Mathematics department who is an instructor in the Computer Science department. Although the two tuples representing that person in STUDENT and INSTRUCTOR will have the same (Name, Ssn) values, they will not agree on the Department value, and so will not be matched. This is because Department has two different meanings in STUDENT (the department where the person studies) and INSTRUCTOR (the department where the person is employed as an instructor).
- If we wanted to apply the OUTER UNION based on the same (Name, Ssn) combination only, we should rename the Department attribute in each table to reflect that they have different meanings and designate them as not being part of the union-compatible attributes. For example, we could rename the attributes as MajorDept in STUDENT and WorkDept in INSTRUCTOR.

6.5 Examples of Queries in Relational Algebra

- Following are additional examples to illustrate the use of the relational algebra operations.
- All examples refer to the database in Figure 5.6.
- In general, the same query can be stated in numerous ways using the various operations. We will state each query in one way and leave it to the reader to come up with equivalent formulations.

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

$$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{Dname='Research'}}(\text{DEPARTMENT})$$

$$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Address}}(\text{RESEARCH\_EMPS})$$

- As a single in-line expression, this query becomes:

$$\pi_{\text{Fname, Lname, Address}} (\sigma_{\text{Dname='Research'}}(\text{DEPARTMENT} \bowtie_{\text{Dnumber=Dno}} (\text{EMPLOYEE}))$$

- This query could be specified in other ways; for example, the order of the JOIN and SELECT operations could be reversed, or the JOIN could be replaced by a NATURAL JOIN after renaming one of the join attributes to match the other join attribute name.

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$$

$$\text{CONTR\_DEPTS} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT})$$

$$\text{PROJ\_DEPT\_MGRS} \leftarrow (\text{CONTR\_DEPTS} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGRS})$$

In this example, we first select the projects located in Stafford, then join them with their controlling departments, and then join the result with the department managers. Finally, we apply a project operation on the desired attributes.

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

$DEPT5\_PROJS(Pno) \leftarrow \pi_{Pnumber}(\sigma_{Dnum=5}(PROJECT))$

$EMP\_PROJ(Ssn, Pno) \leftarrow \pi_{Essn, Pno}(WORKS\_ON)$

$RESULT\_EMP\_SSNS \leftarrow EMP\_PROJ \div DEPT5\_PROJS$

$RESULT \leftarrow \pi_{Lname, Fname}(RESULT\_EMP\_SSNS * EMPLOYEE)$

In this query, we first create a table DEPT5_PROJS that contains the project numbers of all projects controlled by department 5. Then we create a table EMP_PROJ that holds (Ssn, Pno) tuples, and apply the division operation. Notice that we renamed the attributes so that they will be correctly used in the division operation. Finally, we join the result of the division, which holds only Ssn values, with the EMPLOYEE table to retrieve the desired attributes from EMPLOYEE.

Query 4. 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.

$SMITHS(Essn) \leftarrow \pi_{Ssn}(\sigma_{Lname='Smith'}(EMPLOYEE))$

$SMITH\_WORKER\_PROJS \leftarrow \pi_{Pno}(WORKS\_ON * SMITHS)$

$MGRS \leftarrow \pi_{Lname, Dnumber}(EMPLOYEE \bowtie_{Ssn=Mgr\_ssn} DEPARTMENT)$

$SMITH\_MANAGED\_DEPTS(Dnum) \leftarrow \pi_{Dnumber}(\sigma_{Lname='Smith'}(MGRS))$

$SMITH\_MGR\_PROJS(Pno) \leftarrow \pi_{Pnumber}(SMITH\_MANAGED\_DEPTS * PROJECT)$

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

In this query, we retrieved the project numbers for projects that involve an employee named Smith as a worker in SMITH_WORKER_PROJS. Then we retrieved the project numbers for projects that involve an employee named Smith as manager of the department that controls the project in SMITH_MGR_PROJS. Finally, we applied the **UNION** operation on SMITH_WORKER_PROJS and SMITH_MGR_PROJS. As a single in-line expression, this query becomes:

$$\pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn}(\pi_{Ssn}(\sigma_{Lname='Smith'}(EMPLOYEE))) \cup \pi_{Pno}((\pi_{Dnumber}(\sigma_{Lname='Smith'}(\pi_{Lname, Dnumber}(EMPLOYEE))) \bowtie_{Ssn=Mgr\_ssn} DEPARTMENT)) \bowtie_{Dnumber=Dnum} PROJECT)$$

Query 5. List the names of all employees with two or more dependents. Strictly speaking, this query cannot be done in the *basic* (*original*) *relational algebra*.We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* Dependent_name values.

$T1(Ssn, No\_of\_dependents) \leftarrow {}_{Essn}\mathfrak{I}_{COUNT\ Dependent\_name}(DEPENDENT)$

$T2 \leftarrow \sigma_{No\_of\_dependents>2}(T1)$

$RESULT \leftarrow \pi_{Lname, Fname}(T2 * EMPLOYEE)$

Query 6. Retrieve the names of employees who have no dependents. This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

$$ALL\_EMPS \leftarrow \pi_{Ssn}(EMPLOYEE)$$
$$EMPS\_WITH\_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT)$$
$$EMPS\_WITHOUT\_DEPS \leftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$$

We first retrieve a relation with all employee Ssns in ALL_EMPS. Then we create a table with the Ssns of employees who have at least one dependent in EMPS_WITH_DEPS. Then we apply the SET DIFFERENCE operation to retrieve employees Ssns with no dependents in EMPS_WITHOUT_DEPS, and finally join this with EMPLOYEE to retrieve the desired attributes. As a single in-line expression, this query becomes:

$$\pi_{Lname,\ Fname}((\pi_{Ssn}(EMPLOYEE) - \rho_{Ssn}(\pi_{Essn}(DEPENDENT))) * EMPLOYEE)$$

Query 7. List the names of managers who have at least one dependent.

$$MGRS(Ssn) \leftarrow \pi_{Mgr\_ssn}(DEPARTMENT)$$
$$EMPS\_WITH\_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT)$$
$$MGRS\_WITH\_DEPS \leftarrow (MGRS \cap EMPS\_WITH\_DEPS)$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(MGRS\_WITH\_DEPS * EMPLOYEE)$$

In this query, we retrieve the Ssns of managers in MGRs, and the SSNs of employees with at least one dependent in EMPS_WITH_DEPS, then we apply the SET INTERSECTION operation to get the Ssns of managers who have at least one dependent.

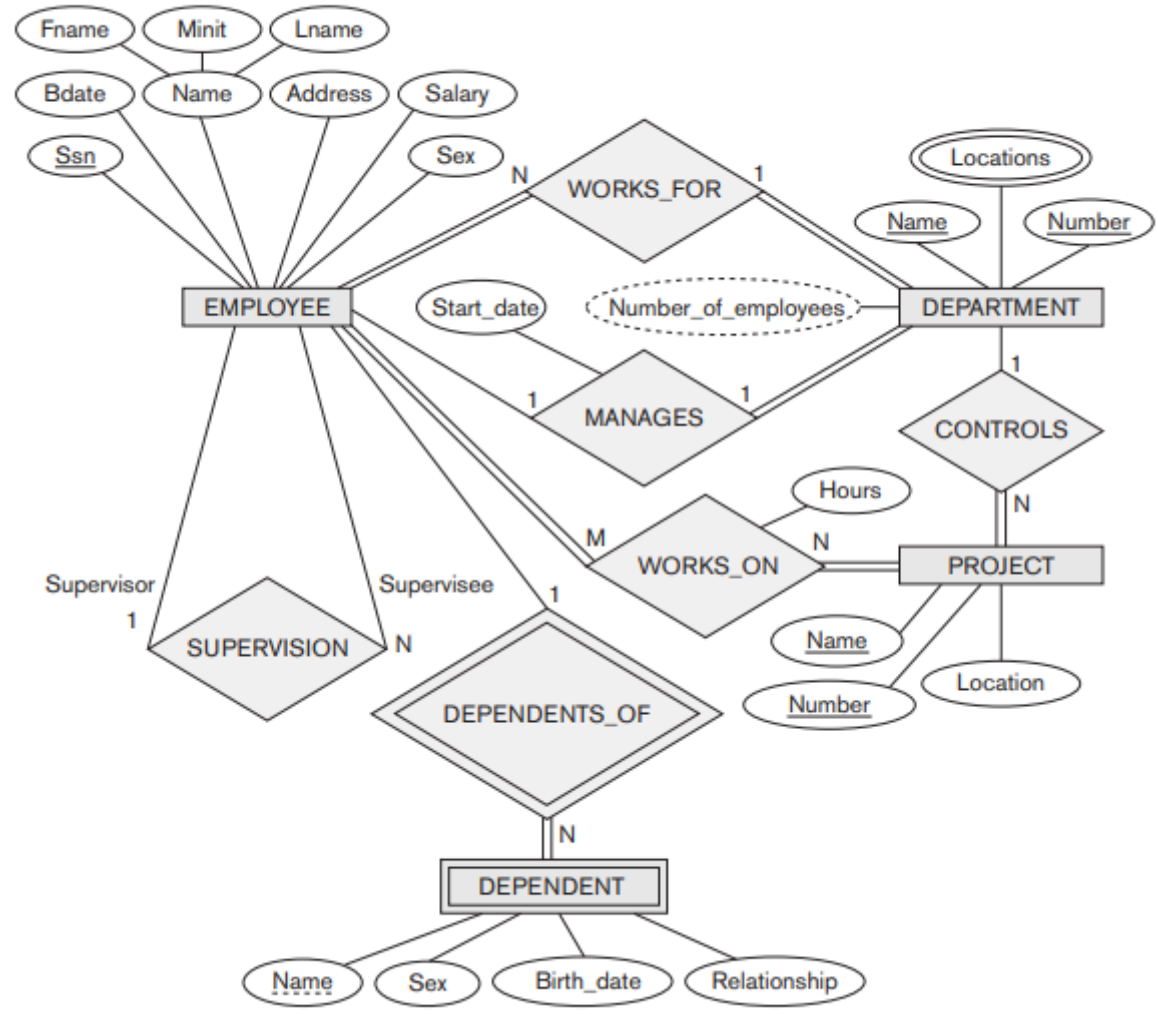
Mapping Conceptual Design into a Logical Design

9.1 Relational Database Design Using ER-to-Relational Mapping

9.1.1 ER-to-Relational Mapping Algorithm

In this section we describe the steps of an algorithm for ER-to-relational mapping. We use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 9.1, and the corresponding COMPANY relational database schema is shown in Figure 9.2 to illustrate the mapping steps. We assume that the mapping will create tables with simple single valued attributes. The relational model constraints defined such as primary keys, unique keys (if any), and referential integrity constraints on the relations, will also be specified in the mapping results.

Figure 9.1
The ER conceptual schema diagram for the COMPANY database.



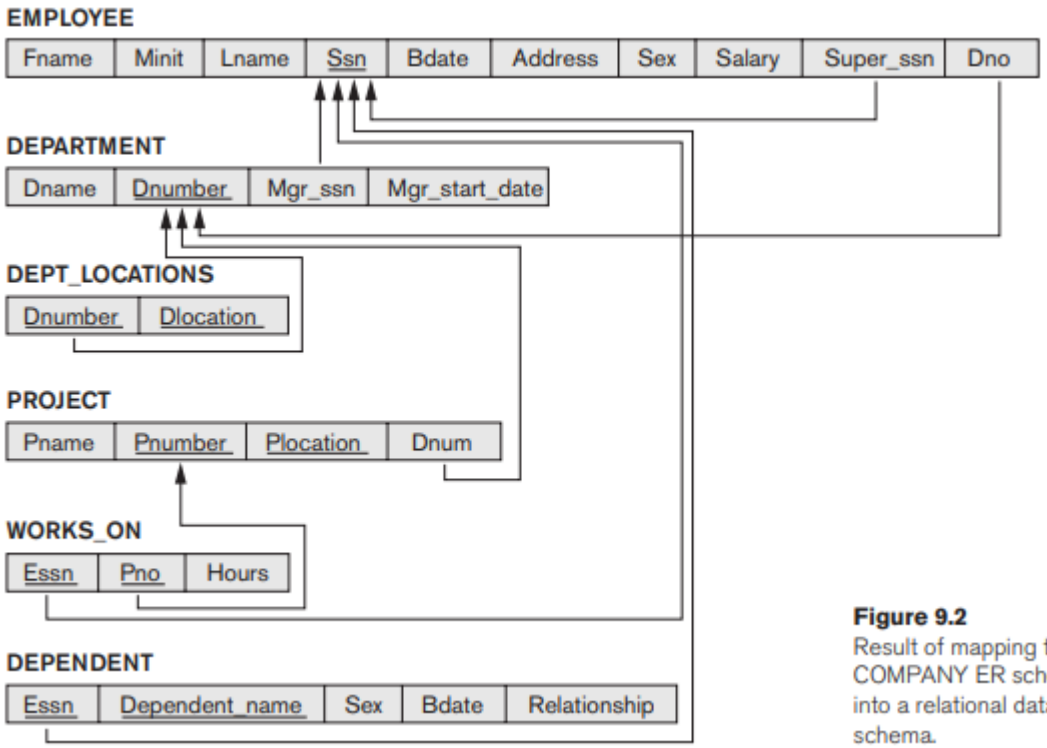


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type
E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R. If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R.
If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify additional (unique) keys of relation R. Knowledge about keys is also kept for indexing purposes and other types of analyses.
In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure 9.2 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT from Figure 9.1. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the attributes Super_ssn and Dno of EMPLOYEE, Mgr_ssn and Mgr_start_date of DEPARTMENT, and Dnum of PROJECT. In our example, we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively. Knowledge that Dname of DEPARTMENT and Pname of PROJECT are unique keys is kept for possible use later in the design.

Step 2: Mapping of Weak Entity Types. For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R. In addition, include as foreign key attributes of R, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any. If there is a weak entity type E2 whose owner is also a weak entity type E1, then E1 should be mapped before E2 to determine its primary key first.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT (see Figure 9.3(b)). We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not necessary. The primary key of the DEPENDENT relation is the combination {Essn, Dependent_name}, because Dependent_name (also renamed from Name in Figure 9.1) is the partial key of DEPENDENT.

Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R.
There are three possible approaches:

(1) the foreign key approach,
(2) the merged relationship approach, and
(3) the cross reference or relationship relation approach.

The first approach is the most useful and should be followed unless special conditions exist, as we discuss below:

1. Foreign key approach: Choose one of the relations—S, say—and include as a foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.

In our example, we map the 1:1 relationship type MANAGES from Figure 9.1 by choosing the participating entity type DEPARTMENT to serve in the role of S because its participation in the MANAGES relationship type is total (every department has a manager). We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it to Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date (see Figure 9.2).

2. Merged relation approach: An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when both participations are total, as this would indicate that the two tables will always have the exact same number of tuples.

3. Cross-reference or relationship relation approach: The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation R is called a relationship relation (or sometimes a lookup table), because each tuple in R represents a relationship instance that relates one tuple from S with one tuple from T. The relation R will include the primary key attributes of S and T as foreign keys to S and T. The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R. The drawback is having an extra relation and requiring extra join operations when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types. There are two possible approaches: (1) the foreign key approach and (2) the cross-reference or relationship relation approach. The first approach is generally preferred as it reduces the number of tables.

1. The foreign key approach: For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the N-side of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S.
To apply this approach to our example, we map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 9.1. For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn. The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation. These foreign keys are shown in Figure 9.2.

2. The relationship relation approach: An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationship. We create a separate relation R whose attributes are the primary keys of S and T, which will also be foreign keys to S and T. The primary key of R is the same as the primary key of S. This option can be used if few tuples in S participate in the relationship to avoid excessive NULL values in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types. In the traditional relational model with no multivalued attributes, the only option for M:N relationships is the relationship relation (cross-reference) option. For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate relationship relation S.
In our example, we map the M:N relationship type WORKS_ON from Figure 9.1 by creating the relation WORKS_ON in Figure 9.2. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively (renaming is not required; it is a design choice). We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}.

Step 6: Mapping of Multivalued Attributes. For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, plus the primary key attribute K—as a foreign key in R—of the relation that repre sents the entity type or relationship type that has A as a multivalued attribute. The primary key of R is the combination of A and K. If the multivalued attribute is com posite, we include its simple components.
In our example, we create a relation DEPT_LOCATIONS (see Figure 9.3(d)). The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, whereas Dnumber—as foreign

key—represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}. A separate tuple will exist in DEPT_LOCATIONS for each loca tion that a department has. It is important to note that in more recent versions of the relational model that allow array data types, the multivalued attribute can be mapped to an array attribute rather than requiring a separate table.

Step 7: Mapping of N-ary Relationship Types. We use the relationship relation option. For each n-ary relationship type R, where n > 2, create a new relationship relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E′ corresponding to E.

Consider the ternary relationship type SUPPLY in Figure 3.17, which relates a SUPPLIER s, PART p, and PROJECT j whenever s is currently supplying p to j; this can be mapped to the relation SUPPLY shown in Figure 9.4, whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

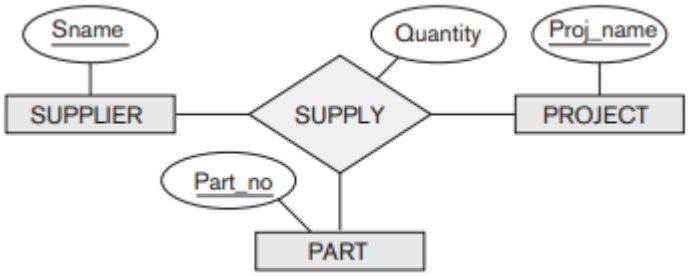
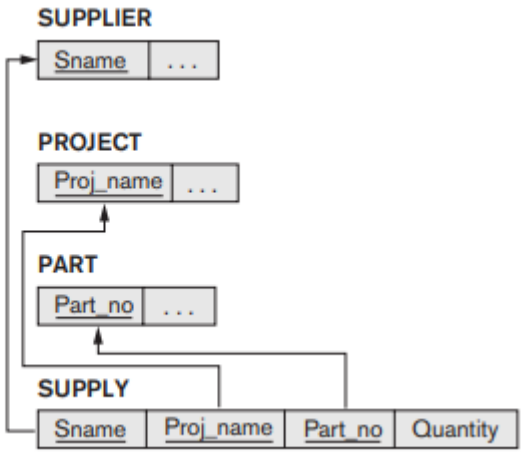


Figure 3.17 The SUPPLY relationship – Ternary Relationship type

Figure 9.4
Mapping the *n*-ary relationship type SUPPLY from Figure 3.17(a).



9.1.2 Discussion and Summary of Mapping for ER Model Constructs

Table 9.1 summarizes the correspondences between ER and relational model constructs and constraints.

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

One of the main points to note in a relational schema, in contrast to an ER schema, is that relationship types are not represented explicitly; instead, they are represented by having two attributes A and B, one a primary key and the other a foreign key (over the same domain) included in two relations S and T. Two tuples in S and T are related when they have the same value for A and B. By using the EQUIJOIN operation (or NATURAL JOIN if the two join attributes have the same name) over S.A and T.B, we can combine all pairs of related tuples from S and T and materialize the relationship. When a binary 1:1 or 1:N relationship type is involved and the foreign key mapping is used, a single join operation is usually needed. When the relationship relation approach is used, such as for a binary M:N relationship type, two join operations are needed, whereas for n-ary relationship types, n joins are needed to fully materialize the relationship instances.