MODULE-11

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.
SQL: SQL data definition and data types, specifying constraints in SQL, retrieval queries in SQL, INSERT, DELETE, and UPDATE statements in SQL, Additional features of SQL.

RELATIONAL MODEL

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.

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.

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. We now define these terms—domain, tuple, attribute, and relatio*n*—formally.

2.1.1 Domains, 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 specify a name for 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. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.
- 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.
- Employee_ages. Possible ages of employees in a company; each must be an integer value between 15 and 80.

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*)*ddd-dddd*, 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 **relation schema** *R*, denoted by *R*(*A*1, *A*2, … , *An*), is made up of a relation name *R* and a list of attributes, *A*1, *A*2, … , *An*. Each **attribute** *Ai* is the name of a role played by some domain *D* in the relation schema *R*. *D* is called the **domain** of *Ai* and is denoted by **dom(*Ai*)**. A relation schema is used to *describe* a relation; *R* is called the **name** of this relation. 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:

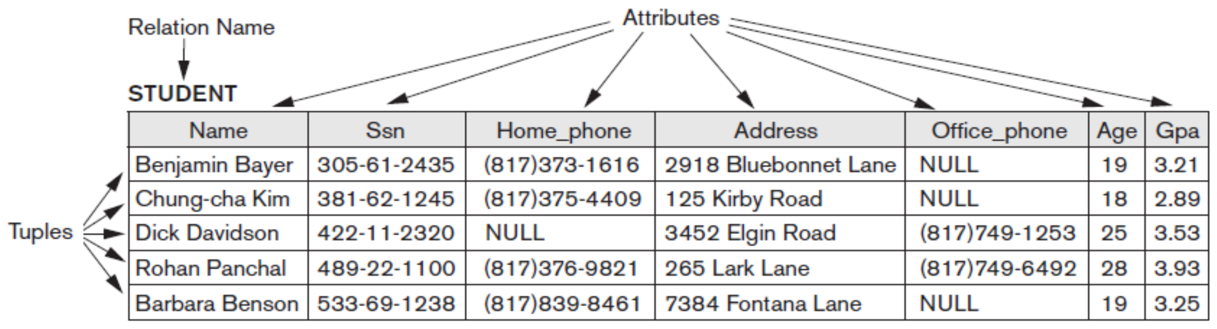
STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

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)

A **relation** (or **relation state**) *r* of the relation schema *R*(*A*1, *A*2, … , *An*), also denoted by *r*(*R*), is a set of *n*-tuples *r* = {*t*1, *t*2, … , *tm*}. Each ***n*-tuple** *t* is an ordered list of *n* values *t* =<*v*1, *v*2, ... , *vn*>, where each value *vi*, $1 \le i \le n$, is an element of dom (*Ai*) or is a special NULL value. The *i*th value in tuple *t*, which corresponds to the attribute *Ai*, is referred to as *t*[*Ai*] or *t*.*Ai* (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 2.1 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 2.1: The attributes and tuples of a relation STUDENT.

The earlier definition of a relation can be *restated* more formally using set theory concepts as follows. A relation (or relation state) *r*(*R*) is a **mathematical relation** of degree *n* on the domains dom(*A*1), dom(*A*2), … , dom(*An*), which is a **subset** of the **Cartesian product** (denoted by ×) of the domains that define *R*:

$$r(R) \subseteq (\text{dom}(A1) \times \text{dom}(A2) \times . . . \times (\text{dom}(An))$$

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

$$|\text{dom}(A1)| \times |\text{dom}(A2)| \times . . . \times |\text{dom}(An)|$$

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. In general, as the state of the real world changes, so does the relation state, by being transformed into another relation state.

2.1.2 Characteristics of Relations

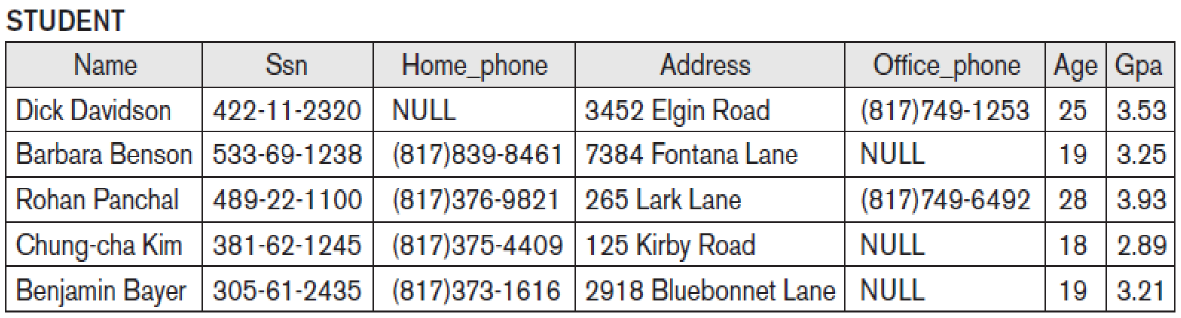
The earlier definition of relations implies certain characteristics that make a relation different from a file or a table. We now discuss some of these characteristics.

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. In other words, a relation is not sensitive to the ordering of tuples.

Example: 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. Many tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 2.1 could be ordered by values of Name, Ssn, Age, or some other attribute. The definition of a relation does not specify any order: There is *no preference* for one ordering over another.

Hence, the relation displayed in Figure 2.2 is considered *identical* to the one shown in Figure 2.1.



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 2.2: The relation STUDENT from Figure 2.1 with a different order of tuples.

Ordering of Values within a Tuple and an Alternative Definition of a Relation. 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.

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, … , *An*} is a *set* of attributes and a relation state *r*(*R*) is a finite set of mappings *r* = {*t*1, *t*2, … , *tm*}, where each tuple *ti* is a **mapping** from *R* to *D*, and *D* is the **union** (denoted by ∪) of the attribute domains; that is, *D* = dom(*A*1) ∪ dom(*A*2) ∪ … ∪ dom(*An*). In this definition, *t*[*Ai*] must be in dom(*Ai*) for 1 ≤ *i* ≤ *n* for each mapping *t* in *r*. Each mapping *ti* 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 *Ai* to a value *vi* from dom(*Ai*). The ordering of attributes is *not* important, because the *attribute name* appears with its *value*. By this definition, the two tuples shown in Figure 2.3 are identical. When the attribute name and value are included together in a tuple, it is known as **self-describing data**, because the description of each value (attribute name) is included in the tuple.

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.

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** and this assumption in mind is called as the **first normal form** assumption.

The 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 2.1, 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 representing foreign students.

Interpretation (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 2.1 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. For example, the first tuple in Figure 2.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

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 2.1. These tuples represent five different propositions or facts in the real world. 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).

2.1.3 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, … , *An*).
- 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, … , *vn*>, where *vi* is the value corresponding to attribute *Ai*. The following notation refers to **component values** of tuples:
    - Both *t*[*Ai*] and *t*.*Ai* (and sometimes *t*[*i*]) refer to the value *vi* in *t* for attribute *Ai*.
    - Both *t*[*Au*, *Aw*, … , *Az*] and *t*.(*Au*, *Aw*, … , *Az*), where *Au*, *Aw*, … , *Az* is a list of attributes from *R*, refer to the subtuple of values <*vu*, *vw*, … , *vz*> from *t* corresponding to the attributes specified in the list.

2.2 Relational Model Constraints and Relational Database Schemas

Relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. 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.

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 the schemas of the data model, typically by specifying them in the DDL. 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 or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

The 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*.

2.2.1 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 other special data types.

2.2.2 Key Constraints and Constraints on NULL Values

A *relation* is defined as a *set of tuples*. By definition, 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.

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:

$$t1[\text{SK}] \neq t2[\text{SK}]$$

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, however, so a more useful concept is that of a *key*, which 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 *uniqueness* property also applies to a superkey.
2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold. This *minimality* property is required for a key but is optional for a superkey.

A superkey may be a key (if it is minimal) or may not be a key (if it is not minimal). Consider the STUDENT relation of Figure 2.1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.8 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 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 2.4 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 2.4.

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 2.4: The CAR relation, with two candidate keys: License_number and Engine_serial_number.

2.2.3 Relational Databases and Relational Database Schemas

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, … , *Rm*} and a set of **integrity constraints** IC. A **relational database state**10 DB of *S* is a set of relation states DB = {*r*1, *r*2, … , *rm*} such that each *ri* is a state of *Ri* and such that the *ri* relation states satisfy the integrity constraints specified in IC. Figure 2.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. In each relation schema, the underlined attribute represents the primary key. Figure 2.6 shows a relational database state corresponding to the COMPANY schema.

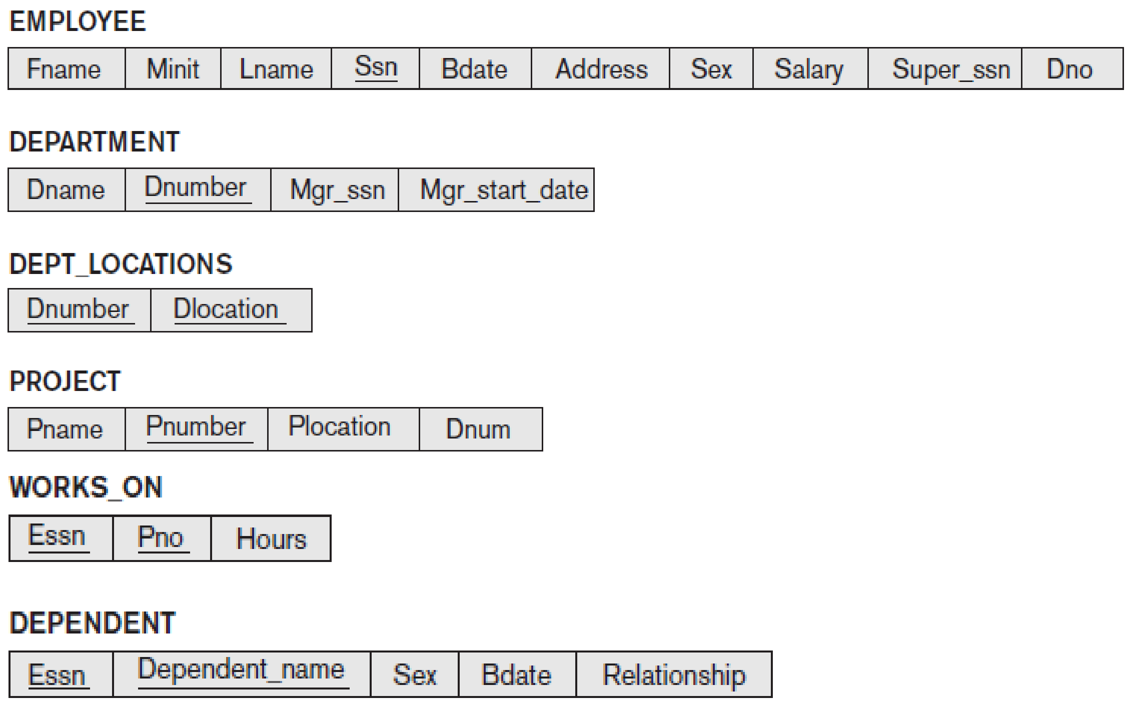
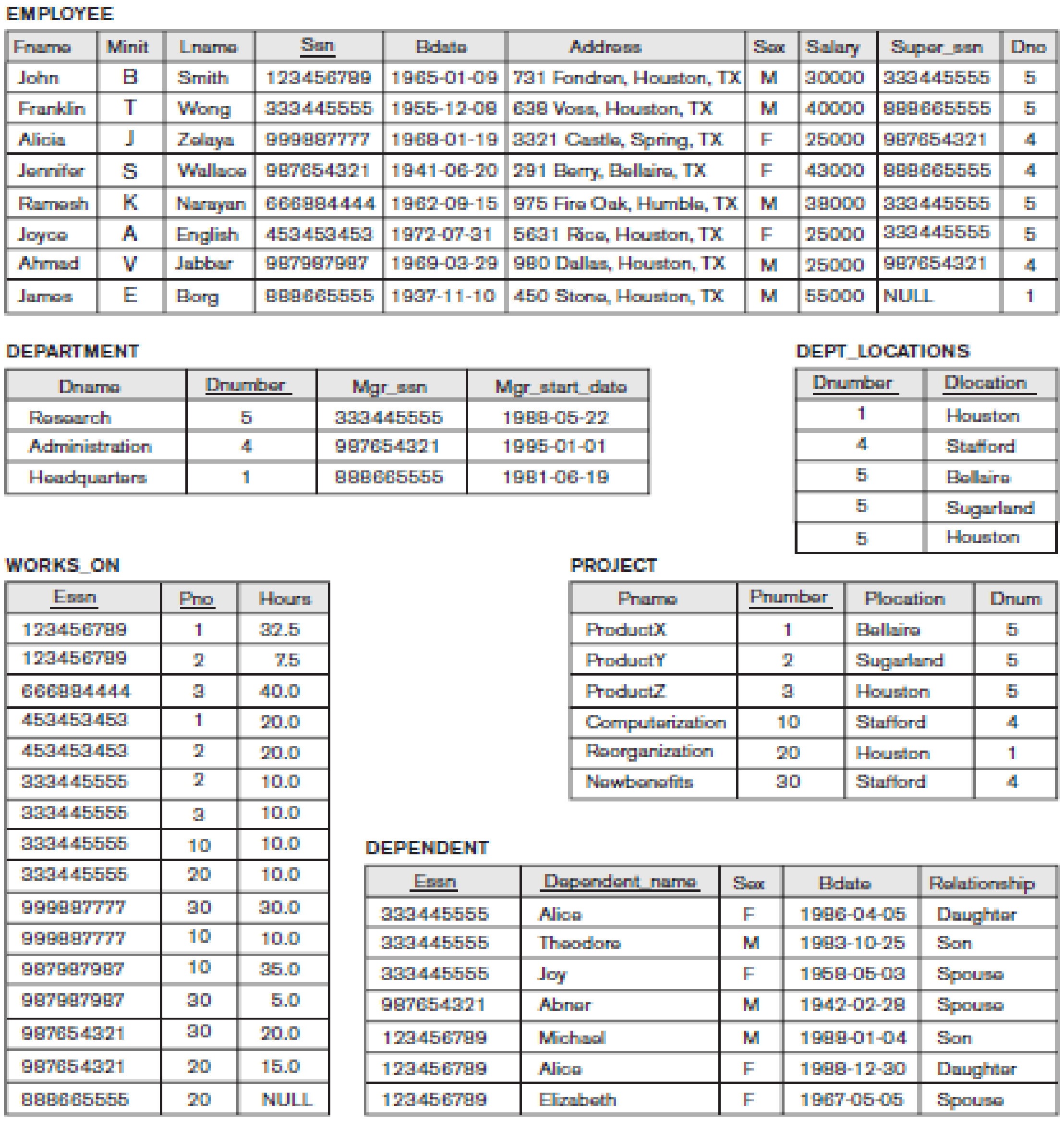


Figure 2.5: Schema diagram for the COMPANY relational database schema.

In Figure 2.5, 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.

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.



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 2.6: One possible database state for the COMPANY relational database schema.

2.2.4 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. For example, in Figure 2.6, 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*. 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.

For example, in Figure 2.6 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 2.6 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 2.7 shows the schema in Figure 2.5 with the referential integrity constraints displayed in this manner.

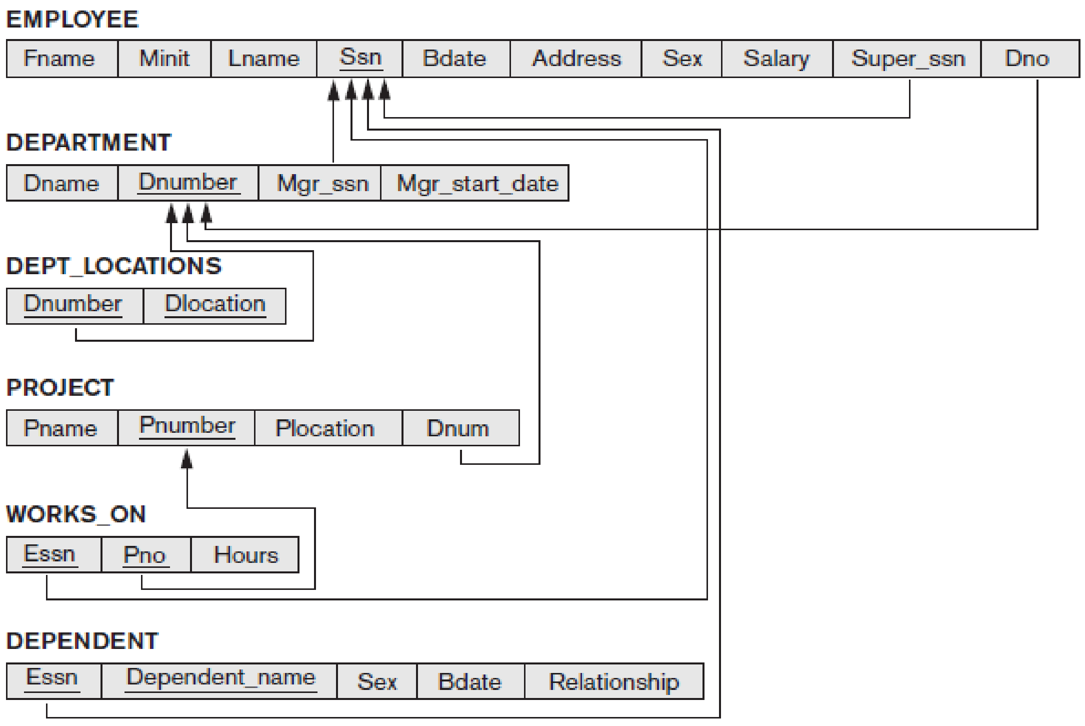


Figure 2.7: Referential integrity constraints displayed on the COMPANY relational database schema.

All integrity constraints should be specified on the relational database schema (that is, specified as part of its definition) if we want the DBMS 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. In SQL, the CREATE TABLE statement of the SQL DDL allows the definition of primary key, unique key, NOT NULL, entity integrity, and referential integrity constraints, among other constraints (see Sections 6.1 and 6.2) .

2.2.5 Other Types of Constraints

The types of constraints are 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.11 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.

2.3 Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates*. 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, respectively. **Insert** is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples.

The database shown in Figure 2.6 for examples and discuss only domain constraints, key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 2.7.

2.3.1 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. 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.

2.3.2 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.

2.3.4 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.

RELATIONAL ALGEBRA

2.4 Unary Relational Operations: SELECT and PROJECT

2.4.1 The SELECT Operation:

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**. SELECT operation can be considered to be a *filter* that keeps only those tuples that satisfy a qualifying condition. 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 and are filtered out.

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:

σDno=4(EMPLOYEE)
σSalary>30000(EMPLOYEE)

In general, the SELECT operation is denoted by

σ<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*.

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

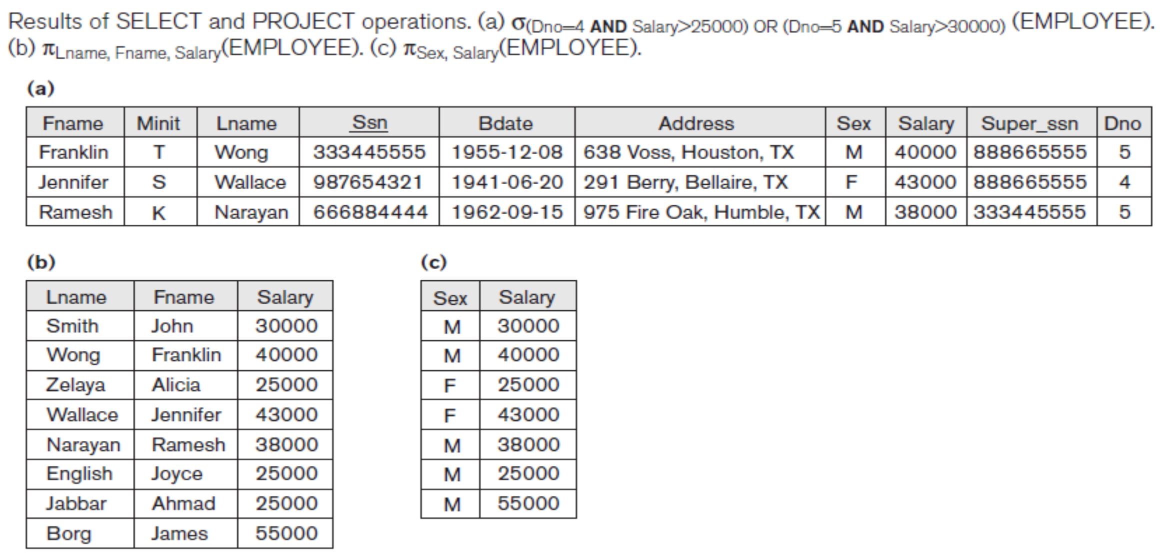
<attribute name> <comparison op> <attribute name>

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:

σ(Dno=4 **AND** Salary>25000) **OR** (Dno=5 **AND** Salary>30000)(EMPLOYEE)

The result is shown in



Results of SELECT and PROJECT operations. (a) $\sigma_{(Dno=4\ \mathbf{AND}\ Salary>25000)\ OR\ (Dno=5\ \mathbf{AND}\ Salary>30000)}$ (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

Figure 2.8: Results of SELECT and PROJECT operations.
(a) σ(Dno=4 **AND** Salary>25000) OR (Dno=5 **AND** Salary>30000) (EMPLOYEE).
(b) πLname, Fname, Salary(EMPLOYEE). (c) πSex, Salary(EMPLOYEE).

The <selection condition> is applied independently to each *individual tuple t* in *R*. This is done by substituting each occurrence of an attribute *Ai* in the selection condition with its value in the tuple *t*[*Ai*]. 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:

- (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (**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,

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

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,

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

In SQL, the SELECT condition is typically specified in the *WHERE clause* of a query.
For example, the following operation:

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

would correspond to the following SQL query:

SELECT *
FROM EMPLOYEE
WHERE Dno=4 **AND** Salary>25000;

2.4.2 The PROJECT Operation

The SELECT operation chooses some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. Therefore, 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 (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*.

For example, consider the following PROJECT operation:

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

The result is shown in Figure 2.8 (c). Notice that the tuple <'F', 25000> appears only once in Figure 2.8 (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.

In SQL, the PROJECT attribute list is specified in the *SELECT clause* of a query. For example, the following operation:

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

would correspond to the following SQL query:

SELECT DISTINCT Sex, Salary
FROM EMPLOYEE

2.4.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 8.1 that depict operation results do not have any names. 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:

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

Figure 2.9 (a) shows the result of this in-line relational algebra expression. The sequence of operations, giving a name to each intermediate relation, and using the **assignment operation,** denoted by ← (left arrow), as follows:

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

(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 2.9 Results of a sequence of operations.
(a) πFname, Lname, Salary (σDno=5(EMPLOYEE)).
(b) Using intermediate relations and renaming of attributes.

To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

TEMP ← σDno=5(EMPLOYEE)
R(First_name, Last_name, Salary) ← πFname, Lname, Salary(TEMP)

These two operations are illustrated in Figure 2.9(b).

The general RENAME operation when applied to a relation *R* of degree *n* is denoted by any of the following three forms:

ρ*S*(*B*1, *B*2, ... , *Bn*)(*R*) or ρ*S*(*R*) or ρ(*B*1, *B*2, ... , *Bn*)(*R*)

where the symbol ρ (rho) is used to denote the RENAME operator, *S* is the new relation name, and *B*1, *B*2, … , *Bn* are the new attribute names.

In SQL, a single query typically represents a complex relational algebra expression. Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

SELECT E.Fname **AS** First_name, E.Lname **AS** Last_name, E.Salary **AS** Salary
FROM EMPLOYEE **AS** E
WHERE E.Dno=5;

2.5 Binary Relational Operations: JOIN and DIVISION

2.5.1 The JOIN Operation:

The **JOIN** operation, denoted by, is used to combine *related tuples* from two relations into single "longer" tuples. 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 2.10. 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. However, JOIN is very important because it is used frequently when specifying database queries. 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:

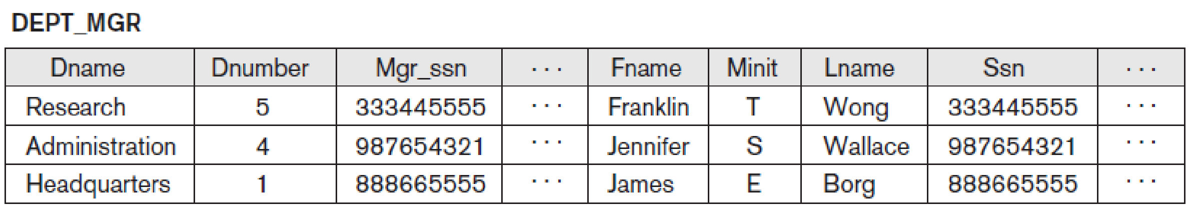
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \text{EMPNAMES Ssn=EssnDEPENDENT}$$

The general form of a JOIN operation on two relations5 $R(A1, A2, \ldots, An)$ and $S(B1, B2, \ldots, Bm)$ is

R <join condition>S

Figure 2.10:

Result of the JOIN operation $\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$.



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	. . .

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**. Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result.

2.5.2 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**. For example, in Figure 2.10, 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.

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 ← PROJECT * ρ(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:

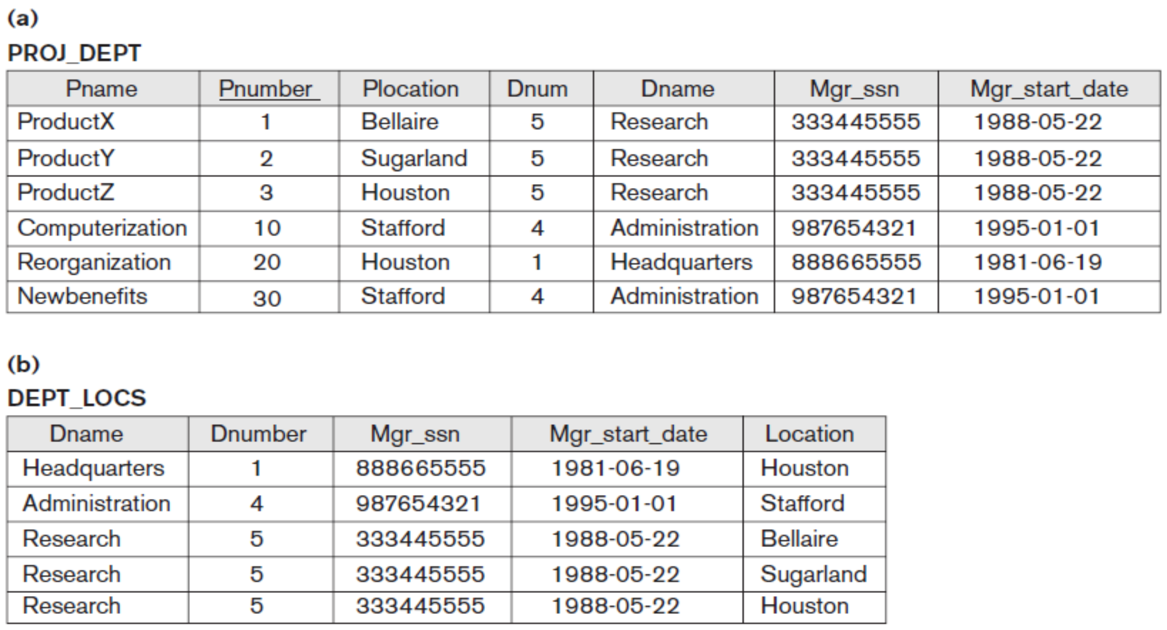
DEPT ← ρ(Dname, Dnum, Mgr_ssn, Mgr_start_date)(DEPARTMENT)
PROJ_DEPT ← PROJECT * DEPT

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 2.11(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 2.11(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)
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 2.11: Results of two natural join operations.
(a) proj_dept ← project * dept.
(b) dept_locs ← department * dept_locations.

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 combines 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 combined information.

2.5.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations 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:

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

JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

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

2.5.4 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:

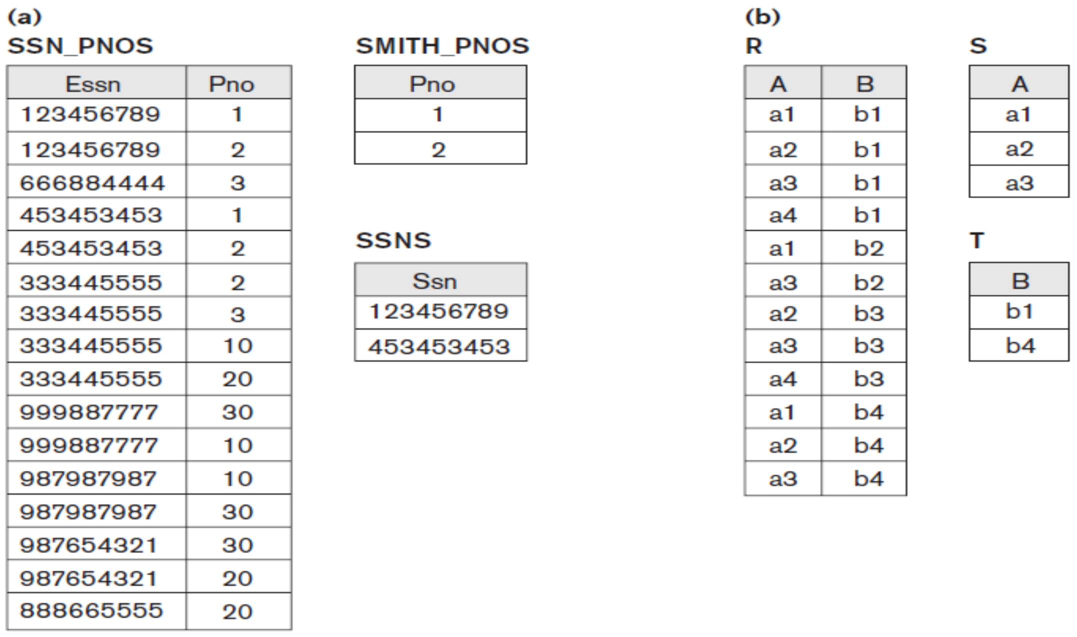
SSN_PNOS ← πEssn, Pno(WORKS_ON)

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

SSNS(Ssn) ← SSN_PNOS ÷ SMITH_PNOS
RESULT ← πFname, Lname(SSNS * EMPLOYEE)

The preceding operations are shown in Figure 2.12(a).

Figure 2.12 (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.



(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

Figure 2.12: The DIVISION operation.
(a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

The DIVISION operation can be expressed as a sequence of π, ×, and – operations as follows:

$T1 \leftarrow \pi_Y(R)$

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

$T \leftarrow T1 - T2$

2.4.5 Notation for Query Trees: 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 2.13 shows a query tree for Query 2 (see Section 6.3.1): *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 and corresponds to the following relational algebra expression:

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

In Figure 2.13, 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 2.13 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.

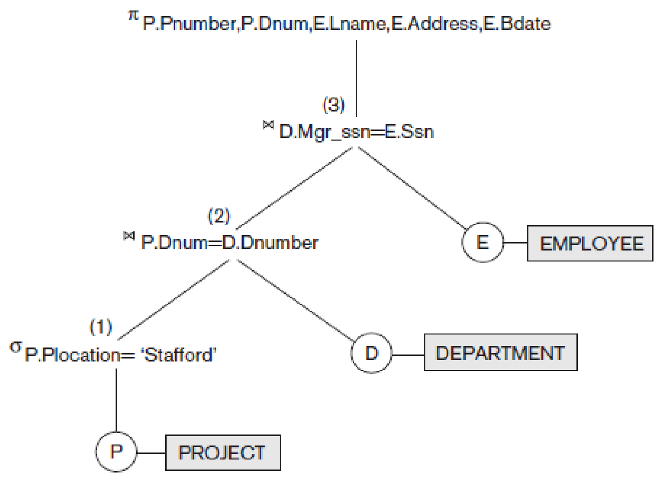


Figure 2.13: Query tree corresponding to the relational algebra expression for Q2.

Table 2.1 lists the various basic relational algebra operations.

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 $R_1 \bowtie_{(<\text{join attributes 1}>),\ (<\text{join attributes 2}>)} R_2$
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 $R_1 *_{(<\text{join attributes 1}>),\ (<\text{join attributes 2}>)} R_2$ 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 $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

2.5 Additional Relational Operations

2.5.1 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:

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

where $F1$, $F2$, … , Fn 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:

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

2.5.2 Aggregate Functions and Grouping

The 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.

We can define an AGGREGATE FUNCTION operation, using the symbol I (pronounced *script F*)7, to specify these types of requests as follows:

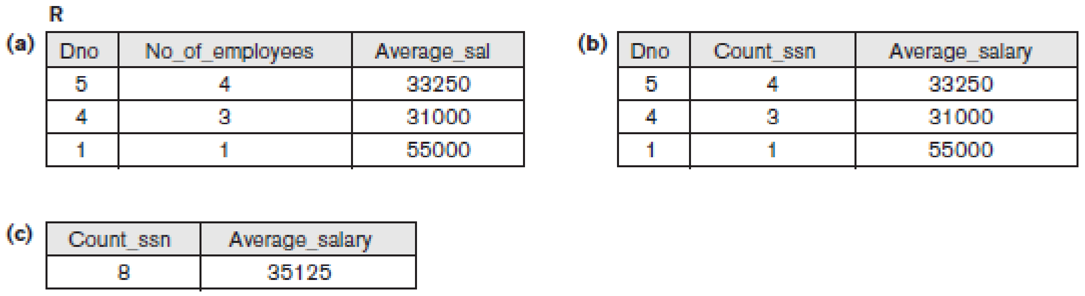
$$_{\text{<grouping attributes>}} \Im_{\text{<function list>}} (R)$$

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:

$$\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})} (_{\text{Dno}} \Im_{\text{COUNT Ssn, AVERAGE Salary}} (\text{EMPLOYEE}))$$

The result of this operation on the EMPLOYEE relation of Figure 2.17 is shown in Figure 2.14(a).



(a) R

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

Figure 2.14:

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. $\Im\ _{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

2.5.3 Recursive Closure Operations

The 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. 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 2.17), we can apply the following operation:

$$\text{BORG\_SSN} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname='James' AND Lname='Borg'}}(\text{EMPLOYEE}))$$
$$\text{SUPERVISION(Ssn1, Ssn2)} \leftarrow \pi_{\text{Ssn,Super\_ssn}}(\text{EMPLOYEE})$$
$$\text{RESULT1(Ssn)} \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{BORG\_SSN})$$

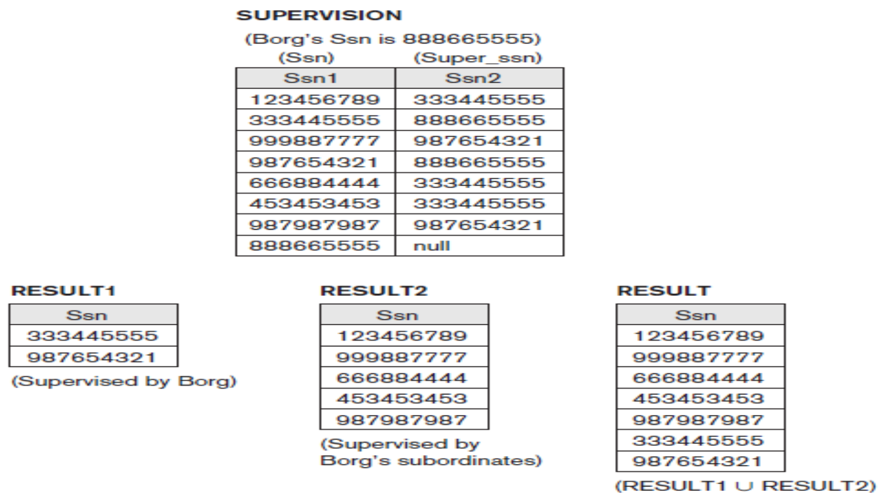
To retrieve all employees supervised by Borg at level 2—that is, all employees *e″* supervised by some employee *e′* who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

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

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:

$$\text{RESULT} \leftarrow \text{RESULT2} \cup \text{RESULT1}$$

The results of these queries are illustrated in Figure 2.15. 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.



SUPERVISION

(Borg's Ssn is 888665555)

Ssn1 (Ssn)	Ssn2 (Super_ssn)
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 2.15: A two-level recursive query.

2.5.4 OUTER JOIN Operations

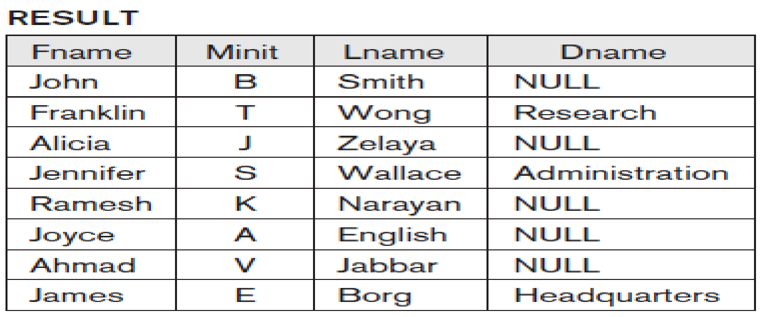
A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in *R*, or all those in *S*, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the 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.

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:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} ⟕_{\text{Ssn=Mgr\_ssn}} \text{DEPARTMENT})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$$

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 2.16.



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

Figure 2.16: The result of a LEFT OUTER JOIN operation.

A similar operation, **RIGHT OUTER JOIN**, denoted by, keeps every tuple in the *second*, or right, relation *S* in the result of *R 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.

2.5.5 The OUTER UNION Operation

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*.

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).

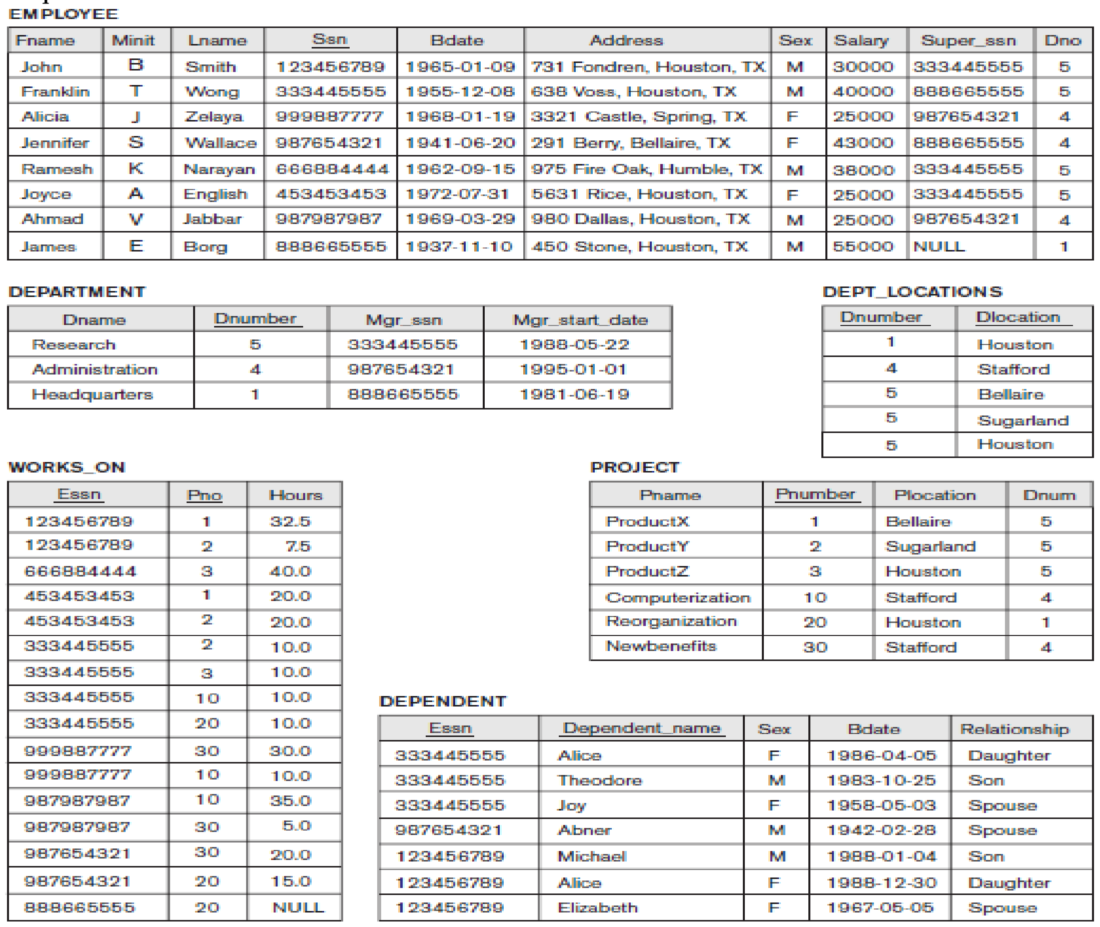
Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department. The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

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.

2.6 Examples of Queries in Relational Algebra

The following are additional examples to illustrate the use of the relational algebra operations. All examples refer to the database in brlow. 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.



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 2.17: Company Relational database schema

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.

$\text{DEPT5\_PROJS} \leftarrow \rho_{(\text{Pno})}(\pi_{\text{Pnumber}}(\sigma_{\text{Dnum}=5}(\text{PROJECT})))$
$\text{EMP\_PROJ} \leftarrow \rho_{(\text{Ssn, Pno})}(\pi_{\text{Essn, Pno}}(\text{WORKS\_ON}))$
$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$
$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$

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 Fname, Lname 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.

$\text{SMITHS(Essn)} \leftarrow \pi_{\text{Ssn}} (\sigma_{\text{Lname='Smith'}}(\text{EMPLOYEE}))$
$\text{SMITH\_WORKER\_PROJS} \leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} * \text{SMITHS})$
$\text{MGRS} \leftarrow \pi_{\text{Lname, Dnumber}}(\text{EMPLOYEE} \bowtie_{\text{Ssn=Mgr\_ssn}} \text{DEPARTMENT})$
$\text{SMITH\_MANAGED\_DEPTS(Dnum)} \leftarrow \pi_{\text{Dnumber}} (\sigma_{\text{Lname='Smith'}}(\text{MGRS}))$
$\text{SMITH\_MGR\_PROJS(Pno)} \leftarrow \pi_{\text{Pnumber}}(\text{SMITH\_MANAGED\_DEPTS} * \text{PROJECT})$
$\text{RESULT} \leftarrow (\text{SMITH\_WORKER\_PROJS} \cup \text{SMITH\_MGR\_PROJS})$

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:

π_{Pno} (WORKS_ON $\bowtie_{\text{Essn=Ssn}}$ (π_{Ssn} ($\sigma_{\text{Lname='Smith'}}$(EMPLOYEE))) $\cup$ π_{Pno} ((π_{Dnumber} ($\sigma_{\text{Lname='Smith'}}$($\pi_{\text{Lname, Dnumber}}$(EMPLOYEE))) $\bowtie_{\text{Ssn=Mgr_ssn}}$DEPARTMENT)) $\bowtie_{\text{Dnum-ber=Dnum}}$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)$\leftarrow$ $_{\text{Essn}}\,\Im\,_{\text{COUNT Dependent\_name}}$(DEPENDENT)
$T2 \leftarrow \sigma_{\text{No\_of\_dependents>2}}(T1)$
RESULT $\leftarrow \pi_{\text{Lname, Fname}}(T2 * \text{EMPLOYEE})$

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 $\leftarrow \pi_{\text{Ssn}}$(EMPLOYEE)
EMPS_WITH_DEPS(Ssn) $\leftarrow \pi_{\text{Essn}}$(DEPENDENT)
EMPS_WITHOUT_DEPS $\leftarrow$ (ALL_EMPS – EMPS_WITH_DEPS)
RESULT $\leftarrow \pi_{\text{Lname, Fname}}$(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:

$\pi_{\text{Lname, Fname}}$(($\pi_{\text{Ssn}}$(EMPLOYEE) – ρ_{Ssn}(π_{Essn}(DEPENDENT))) * EMPLOYEE)

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

MGRS(Ssn) $\leftarrow \pi_{\text{Mgr\_ssn}}$(DEPARTMENT)
EMPS_WITH_DEPS(Ssn) $\leftarrow \pi_{\text{Essn}}$(DEPENDENT)
MGRS_WITH_DEPS $\leftarrow$ (MGRS $\cap$ EMPS_WITH_DEPS)
RESULT $\leftarrow \pi_{\text{Lname, Fname}}$(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.

2.7 Mapping Conceptual Design into a Logical Design

2.7.1 Relational Database Design using ER-to-Relational mapping.

9.1.1 ER-to-Relational Mapping Algorithm

The steps of an algorithm for ER-to-relational mapping. Consider an example of COMPANY database to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 2.18, 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.

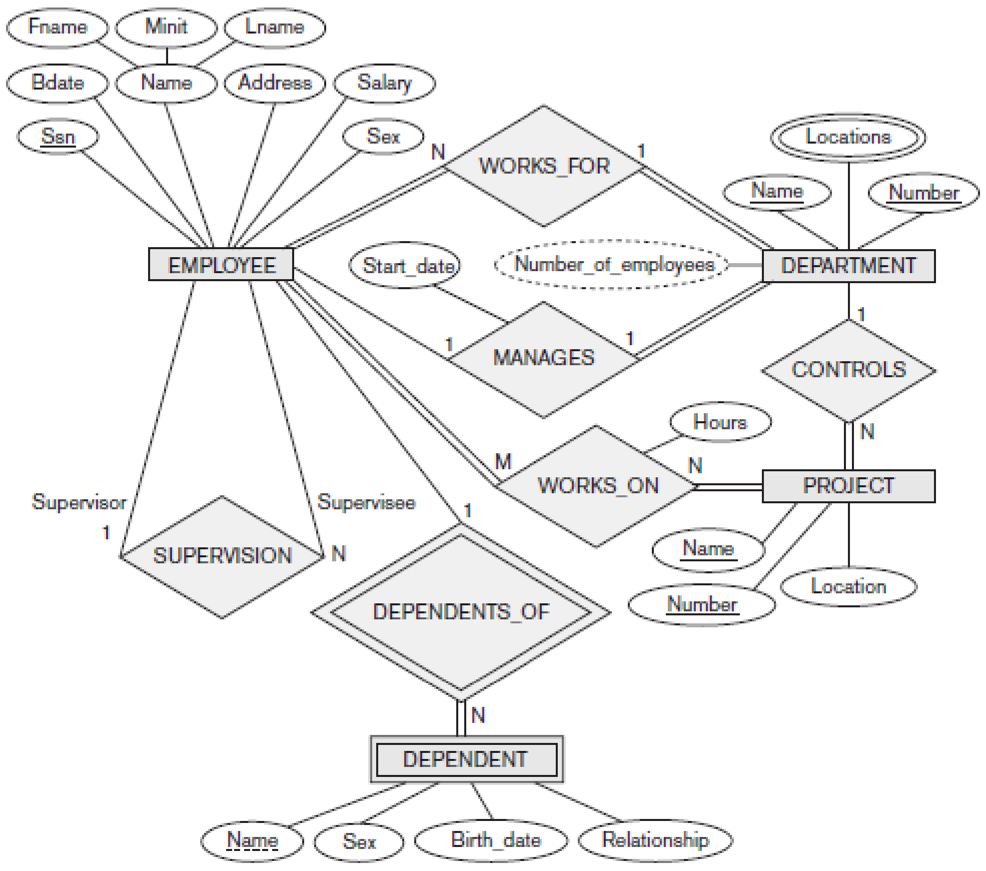


Figure 2.18: The ER conceptual schema diagram for the COMPANY database.

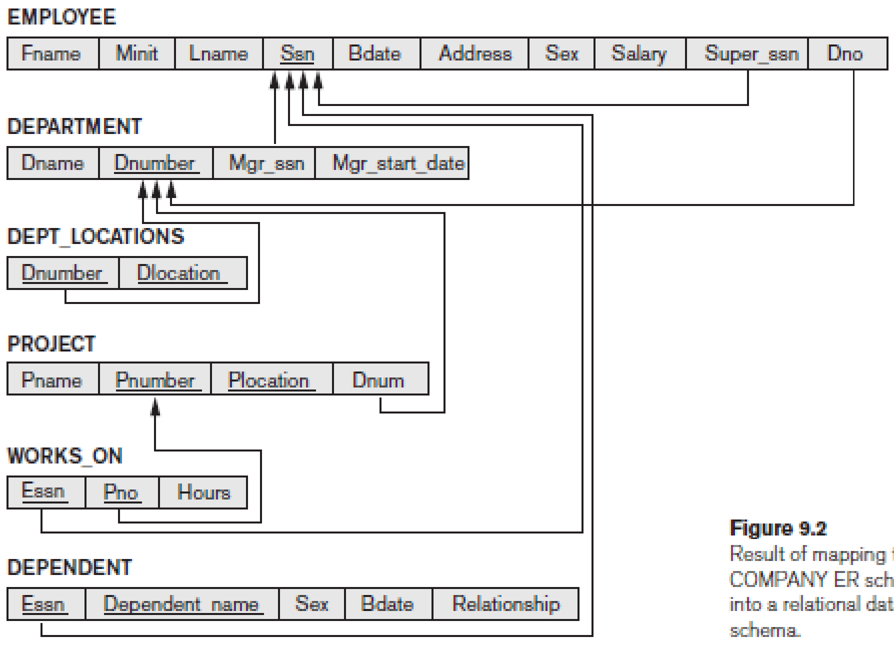


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

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*. 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 2.18. 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.

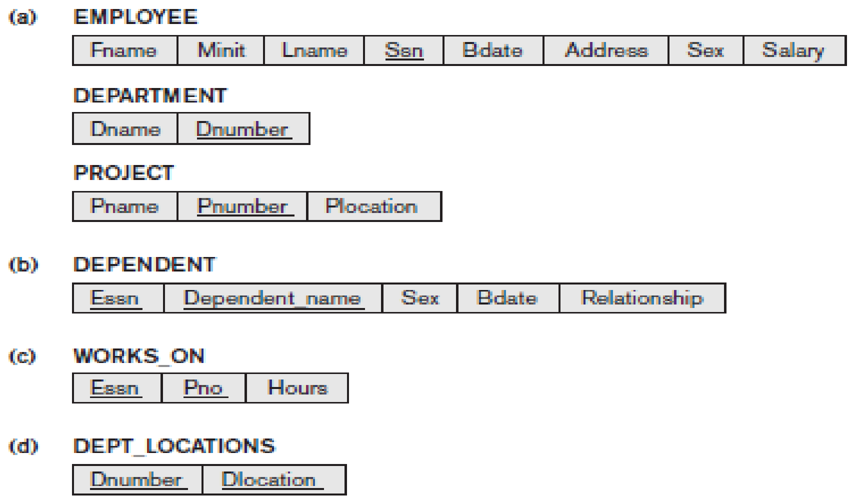


Figure 2.19: Illustration of some mapping steps.
(a) *Entity* relations after step 1.
(b) Additional *weak entity* relation after step 2.
(c) *Relationship* relations after step 5.
(d) Relation representing multivalued attribute after step 6.

The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple represents an entity instance. The result after this mapping step is shown in Figure 2.19(a).

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 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 *E*2 whose owner is also a weak entity type *E*1, then *E*1 should be mapped before *E*2 to determine its primary key first.

Create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT (see Figure 2.19(b)). 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 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 of the 1:1 relationship type *R* as attributes of *S*.
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 have the exact same number of tuples at all times.
3. **Cross-reference or relationship relation approach: T**his approach is required for binary M:N relationships. The relation *R* is called a **relationship relation** 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.
2. **The relationship relation approach:** 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 because of the M:N cardinality ratio; we must create a separate *relationship relation S*.

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 represents the entity type or relationship type that has *A* as a multivalued attribute.

Step 7: Mapping of *N*-ary Relationship Types. 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.

SQL

2.8 SQL data definition and data types: SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), types, and domains, as well as other constructs such as views, assertions, and triggers.

2.8.1 Schema and Catalog Concepts in SQL

An **SQL schema** is identified by a **schema name** and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema. Schema **elements** include tables, types, constraints, views, domains, and other constructs that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. For example, the following statement creates a schema called COMPANY owned by the user with authorization identifier 'Jsmith'. Note that each statement in SQL ends with a semicolon.

CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmith';

2.8.2 The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and possibly attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared.

The SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. For example, by writing

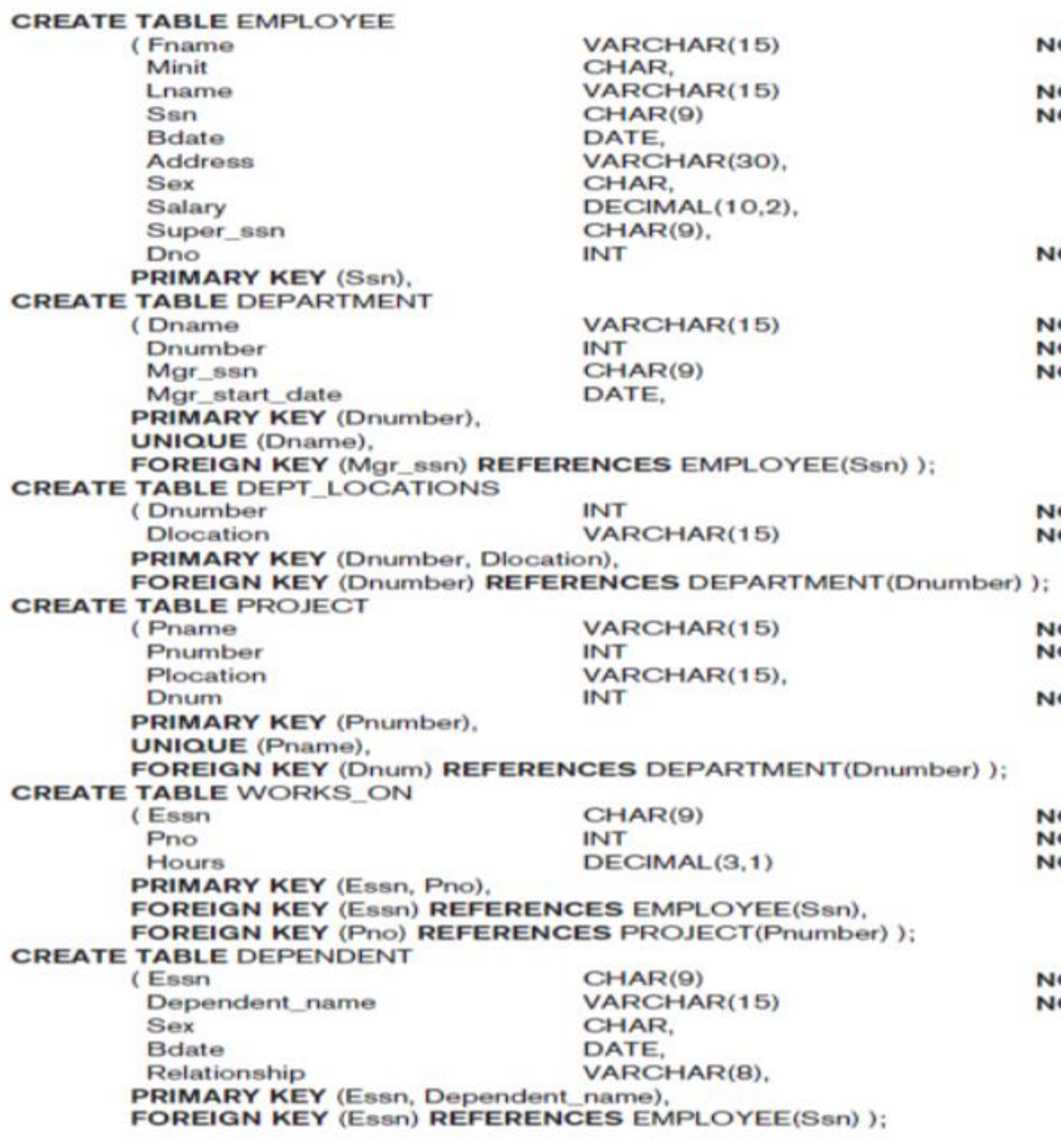
CREATE TABLE COMPANY.EMPLOYEE

rather than

CREATE TABLE EMPLOYEE

The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the table and its rows are actually created and stored as a file by the DBMS. In SQL, the attributes in a base table are considered to be *ordered in the sequence in which they are specified* in the CREATE TABLE statement.

Figure 2.20 show the complete COMPANY Schema.



```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Figure 2.20: SQL CREATE TABLE data definition statements for defining the COMPANY schema

2.8.3 Attribute Data Types and Domains in SQL

The basic **data types** available for attributes include numeric, character string, bit string, Boolean, date, and time.

1. **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(*i*, *j*)—or DEC(*i*, *j*) or NUMERIC(*i*, *j*)—where *i*, the *precision*, is the total number of decimal digits and *j*, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.
2. **Character-string** data types are either fixed length—CHAR(*n*) or CHARACTER(*n*), where *n* is the number of characters—or varying length— VARCHAR(*n*) or CHAR VARYING(*n*) or CHARACTER VARYING(*n*), where *n* is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive* (a distinction is made between uppercase and lowercase).
3. **Bit-string** data types are either of fixed length *n*—BIT(*n*)—or varying length— BIT VARYING(*n*), where *n* is the maximum number of bits. The default for *n*, the length of a character string or bit string, is 1.

4. A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.
5. The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM: SS. Only valid dates and times should be allowed by the SQL implementation.

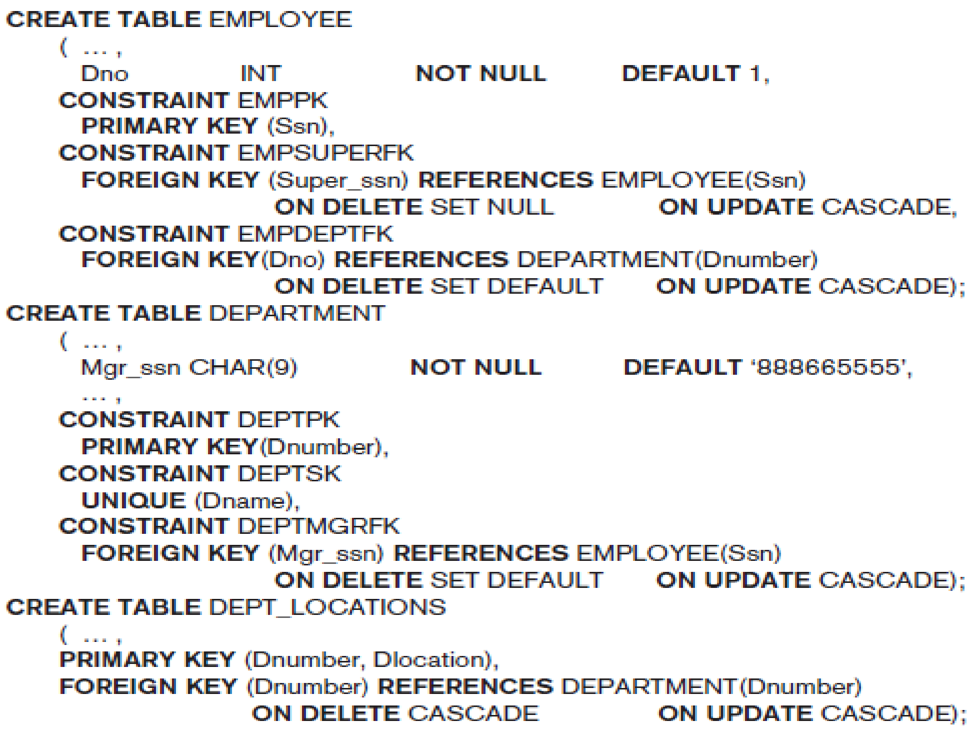
Some additional data types are discussed below. The list of types discussed here is not exhaustive; different implementations have added more data types to SQL.

1. A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single-quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2014-09-27 09:12:47.648302'.
2. Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

2.9 Specifying Constraints in SQL

This includes key and referential integrity constraints, restrictions on attribute domains and NULLs, and constraints on individual tuples within a relation using the CHECK clause.

2.9.1 Specifying Attribute Constraints and Attribute Defaults: Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the *primary key* of each relation, but it can be specified for any other attributes whose values are required not to be NULL, as shown in Figure 2.20.



```
CREATE TABLE EMPLOYEE
    ( ... ,
      Dno          INT            NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                       ON DELETE SET NULL          ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                       ON DELETE SET DEFAULT      ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
    ( ... ,
      Mgr_ssn CHAR(9)             NOT NULL       DEFAULT '888665555',
      ... ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                       ON DELETE SET DEFAULT      ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
    ( ... ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                      ON DELETE CASCADE           ON UPDATE CASCADE);
```

Figure 2.21: Example illustrating how default attributes values and referential integrity triggered actions are specified in SQL.

It is also possible to define a *default value* for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute. Figure 2.21 illustrates an example of specifying a default manager for a new department and a default department for a new employee. If no default clause is specified, the default *default value* is NULL for attributes *that do not have* the NOT NULL constraint.

For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

Dnumber INT **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

CREATE DOMAIN D_NUM **AS** INTEGER
CHECK (D_NUM > 0 **AND** D_NUM < 21);

2.9.2 Specifying Key and Referential Integrity Constraints

The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a *single* attribute, the clause can follow the attribute directly. For example, the primary key of DEPARTMENT can be specified as follows

Dnumber INT **PRIMARY KEY**,

The **UNIQUE** clause specifies alternate (unique) keys, also known as candidate keys as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 2.20. The **UNIQUE** clause can also be specified directly for a unique key if it is a single attribute, as in the following example:

Dname VARCHAR (15) **UNIQUE**,

Referential integrity is specified via the **FOREIGN KEY** clause, as shown in Figure 2.20
In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples. It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema.

2.9.3 Giving Names to Constraints

Figure 2.21 also illustrates how a constraint may be given a **constraint name**, following the keyword **CONSTRAINT**. The names of all constraints within a particular schema must be unique.

2.9.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **row-based** constraints because they apply to each row *individually* and are checked whenever a row is inserted or modified.

For example, suppose that the DEPARTMENT table in Figure 2.20 had an additional attribute Dept_create_date, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date.

CHECK (Dept_create_date <= Mgr_start_date);

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL.

2.10 Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT** statement. The SELECT statement *is not the same as* the SELECT operation of relational algebra. We will use example queries specified on the schema of Figure 2.5 and will refer to the sample database state shown in Figure 2.6 to show the results of some of these queries. In this section, we present the features of SQL for *simple retrieval queries*.

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

Queries in SQL can be very complex. We will start with simple queries, and then progress to more complex ones in a step-by-step manner. The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
SELECT <attribute list>
FROM <table list>
WHERE <condition>;
```

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q 0:    SELECT Bdate, Address
        FROM EMPLOYEE
        WHERE Fname = 'John' AND Minit = 'B' AND Lname = 'Smith';
```

This query involves only the EMPLOYEE relation listed in the FROM clause. The query *selects* the individual EMPLOYEE tuples that satisfy the condition of the WHERE clause, then *projects* the result on the Bdate and Address attributes listed in the SELECT clause.

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

```
Q 1:    SELECT Fname, Lname, Address
        FROM EMPLOYEE, DEPARTMENT
        WHERE Dname = 'Research' AND Dnumber = Dno;
```

In the WHERE clause of Q1, the condition Dname = 'Research' is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE.

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.

```
Q 2:    SELECT Pnumber, Dnum, Lname, Address, Bdate
        FROM PROJECT, DEPARTMENT, EMPLOYEE
        WHERE Dnum = Dnumber AND Mgr_ssn = Ssn AND Plocation = 'Stafford'
```

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn = Ssn relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a *combination* of one project, one department (that controls the project), and one employee (that manages the department). The projection attributes are used to choose the attributes to be displayed from each combined tuple.

2.10.2 Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in *different tables.* If this is the case, and a multitable query refers to two or more attributes with the same name, we *must* **qualify** the attribute name with the relation name to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period.

To illustrate this, suppose that in Figures 2.5 and 2.6 the Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes Name and Dnumber in Q1A to specify which ones we are referring to, because the same attribute names are used in both relations:

```
Q1A:    SELECT Fname, EMPLOYEE.Name, Address
        FROM EMPLOYEE, DEPARTMENT
        WHERE DEPARTMENT.Name = 'Research' AND
                    DEPARTMENT.Dnumber = EMPLOYEE.Dnumber;
```

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. Q1 can be rewritten as Q1′ below with fully qualified attribute names. We can also rename the table names to shorter names by creating an *alias* for each table name to avoid repeated typing of long table names.

```
Q1′:    SELECT EMPLOYEE.Fname, EMPLOYEE.LName, EMPLOYEE.Address
        FROM EMPLOYEE, DEPARTMENT
        WHERE DEPARTMENT.DName = 'Research' AND
                        DEPARTMENT.Dnumber = EMPLOYEE.Dno;
```

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice, as in the following example.

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8:     SELECT E.Fname, E.Lname, S.Fname, S.Lname
        FROM EMPLOYEE AS E, EMPLOYEE AS S
        WHERE E.Super_ssn = S.Ssn;
```

In this case, we are required to declare alternative relation names E and S, called **aliases** or **tuple variables**, for the EMPLOYEE relation. An alias can follow the keyword **AS**, as shown in Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q8. It is also possible to **rename** the relation attributes within the query in SQL by giving them aliases. For example, if we write

EMPLOYEE **AS** E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)

in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on.

We can use this alias-naming or **renaming** mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once. In fact, this practice is recommended since it results in queries that are easier to comprehend. For example, we could specify query Q1 as in Q1B:

```
Q1B: SELECT E.Fname, E.LName, E.Address
     FROM EMPLOYEE AS E, DEPARTMENT AS D
     WHERE D.DName = 'Research' AND D.Dnumber = E.Dno;
```

2.10.3 Unspecified WHERE Clause and Use of the Asterisk

A *missing* WHERE clause indicates no condition on tuple selection; hence, *all tuples* of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected. For example, Query 9 selects all EMPLOYEE Ssns and Query 10 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not.

Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:  SELECT Ssn
     FROM EMPLOYEE;

Q10: SELECT Ssn, Dname
     FROM EMPLOYEE, DEPARTMENT;
```

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result. Notice that Q10 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra. If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the actual CROSS PRODUCT.

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes* for example, EMPLOYEE.* refers to all attributes of the EMPLOYEE table.

Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 6.3(g)), query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department, and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
Q1C: SELECT *
     FROM EMPLOYEE
     WHERE Dno = 5;
```

Q1D: **SELECT** *
FROM EMPLOYEE, DEPARTMENT
WHERE Dname = 'Research' **AND** Dno = Dnumber;

Q10A: **SELECT** *
FROM EMPLOYEE, DEPARTMENT;

2.10.4 Tables as Sets in SQL

SQL usually treats a table not as a set but rather as a **multiset**; *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple. If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither ALL nor DISTINCT—as in our previous examples—is equivalent to SELECT ALL.

For example, Q11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query. If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword **DISTINCT** as in Q11A.

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

Q11: **SELECT ALL** Salary
FROM EMPLOYEE;
Q11A: **SELECT DISTINCT** Salary
FROM EMPLOYEE;

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra. There are set union (**UNION**), set difference (**EXCEPT**), and set intersection (**INTERSECT**) operations. The next example illustrates the use of UNION.

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

Q4A: (**SELECT DISTINCT** Pnumber
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum = Dnumber **AND** Mgr_ssn = Ssn **AND** Lname = 'Smith')
UNION
(**SELECT DISTINCT** Pnumber
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber = Pno **AND** Essn = Ssn **AND** Lname = 'Smith');

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project. Notice that if several employees have the last name 'Smith', the project names involving any of them will be retrieved.

Applying the UNION operation to the two SELECT queries gives the desired result. SQL also has corresponding multiset operations, which are followed by the keyword **ALL** (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated).

2.10.5 Substring Pattern Matching and Arithmetic Operators

The first feature allows comparison conditions on only parts of a character string, using the **LIKE** comparison operator. This can be used for string **pattern matching**. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character.

For example, consider the following query.

Query 12. Retrieve all employees whose address is in Houston, Texas.

```
Q12: SELECT Fname, Lname
FROM EMPLOYEE
WHERE Address LIKE '%Houston,TX%';
```

To retrieve all employees who were born during the 1970s, we can use Query Q12A. Here, '7' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a placeholder for an arbitrary character.

Query 12A. Find all employees who were born during the 1950s.

```
Q12:  SELECT Fname, Lname
      FROM EMPLOYEE
      WHERE Bdate LIKE '_ _ 7 _ _ _ _ _ _ _';
```

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character*, which is specified after the string using the keyword ESCAPE. For example, 'AB_CD\%EF' ESCAPE '\' represents the literal string 'AB_CD%EF' because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (' ') if they are to be included in a string because they are used to begin and end strings. If an apostrophe (') is needed, it is represented as two consecutive apostrophes (") so that it will not be interpreted as ending the string.

The standard arithmetic operators for addition (+), subtraction (−), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10% raise; we can issue Query 13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

Query 13. Show the resulting salaries if every employee working on the 'ProductX' project is given a 10% raise.

```
Q13:  SELECT E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
      FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
      WHERE E.Ssn = W.Essn AND W.Pno = P.Pnumber AND
                                      P.Pname = 'ProductX';
```

For string data types, the concatenate operator || can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (−) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator, which can be used for convenience, is **BETWEEN**, which is illustrated in Query 14.

Query 14. Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT *
      FROM EMPLOYEE
      WHERE (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

The condition (Salary **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the condition ((Salary >= 30000) **AND** (Salary <= 40000)).

2.10.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause. This is illustrated by Query 15.

Query 15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
Q15:  SELECT D.Dname, E.Lname, E.Fname, P.Pname
      FROM DEPARTMENT AS D, EMPLOYEE AS E, WORKS_ON AS W, PROJECT
      AS P
      WHERE D.Dnumber = E.Dno AND E.Ssn = W.Essn AND W.Pno = P.Pnumber
      ORDER BY D.Dname, E.Lname, E.Fname;
```

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly. For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause of Q15 can be written as **ORDER BY** D.Dname **DESC**, E.Lname **ASC**, E.Fname **ASC**

2.10.7 Discussion and Summary of Basic SQL Retrieval Queries

A *simple* retrieval query in SQL can consist of up to four clauses, but only the first two—SELECT and FROM—are mandatory. The clauses are specified in the following order, with the clauses between square brackets […] being optional:

```
SELECT <attribute list>
FROM <table list>
[ WHERE <condition> ]
[ ORDER BY <attribute list> ];
```

The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including join conditions if needed. ORDER BY specifies an order for displaying the results of a query.

2.11 INSERT, DELETE, and UPDATE Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

2.11.1 The INSERT Command

INSERT is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation shown in Figure 2.5 and specified in the CREATE TABLE EMPLOYEE … command, we can use U1:

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

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification *and* no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be *left out*. For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes, we can use U1A:

```
U1A: INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn)
     VALUES ('Richard', 'Marini', 4, '653298653');
```

Attributes not specified in U1A are set to their DEFAULT or to NULL, and the values are listed in the same order as the *attributes are listed in the INSERT* command itself. It is also possible to insert into a relation *multiple tuples* separated by commas in a single INSERT command. The attribute values forming *each tuple* are enclosed in parentheses.

For example, if we issue the command in U2 on the database shown in Figure 2.6, the DBMS should *reject* the operation because no DEPARTMENT tuple exists in the database with Dnumber = 2. Similarly, U2A would be *rejected* because no Ssn value is provided and it is the primary key, which cannot be NULL.

```
U2:  INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno)
     VALUES ('Robert', 'Hatcher', '980760540', 2);
```

(U2 is rejected if referential integrity checking is provided by DBMS.)

```
U2A: INSERT INTO EMPLOYEE (Fname, Lname, Dno)
     VALUES ('Robert', 'Hatcher', 5);
```

(U2A is rejected if NOT NULL checking is provided by DBMS.)

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query*. For example, to create a temporary table that has the employee last name, project name, and hours per week for each employee working on a project, we can write the statements in U3A and U3B:

U3A: CREATE TABLE WORKS_ON_INFO(Emp_name VARCHAR(15), Proj_name VARCHAR(15), Hours_per_week DECIMAL(3,1));

```
U3B:  INSERT INTO WORKS_ON_INFO ( Emp_name, Proj_name,    Hours_per_week )
      SELECT E.Lname, P.Pname, W.Hours
      FROM PROJECT P, WORKS_ON W, EMPLOYEE E
      WHERE P.Pnumber = W.Pno AND W.Essn = E.Ssn;
```

A table WORKS_ON_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B.

The variation for loading data is to create a new table TNEW that has the same attributes as an existing table T, and load some of the data currently in T into TNEW. The syntax for doing this uses the LIKE clause. For example, if we want to create a table D5EMPS with a similar structure to the EMPLOYEE table and load it with the rows of employees who work in department 5, we can write the following SQL:

```
CREATE TABLE D5EMPS LIKE EMPLOYEE
(SELECT E.*
FROM EMPLOYEE AS E
WHERE E.Dno = 5) WITH DATA;
```

The clause WITH DATA specifies that the table will be created and loaded with the data specified in the query, although in some implementations it may be left out.

2.11.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition. The DELETE commands in U4A to U4D, if applied independently to the database state shown in Figure 2.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A:  DELETE FROM EMPLOYEE
      WHERE Lname = ‘Brown’;
U4B:  DELETE FROM EMPLOYEE
      WHERE Ssn = ‘123456789’;
U4C:  DELETE FROM EMPLOYEE
      WHERE Dno = 5;
U4D:  DELETE FROM EMPLOYEE;
```

2.11.3 The UPDATE Command: The **UPDATE** command is used to modify attribute values of one or more selected tuples. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity constraints of the DDL. An additional **SET** clause in the UPDATE command specifies the attributes to be modified and their new values.

For example, to change the location and controlling department number of project number 10 to ‘Bellaire’ and 5, respectively, we use U5:

```
U5:   UPDATE PROJECT
      SET Plocation = ‘Bellaire’, Dnum = 5
      WHERE Pnumber = 10;
```

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10% raise in salary, as shown in U6. In this request, the modified Salary value depends on the original Salary value in each tuple, so two references to the Salary attribute are needed. In the SET clause, the reference to the Salary attribute on the right refers to the old Salary value *before modification,* and the one on the left refers to the new Salary value *after modification*:

U6: **UPDATE** EMPLOYEE
SET Salary = Salary * 1.1
WHERE Dno = 5;

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

2.12 Additional Features of SQL

SQL has a number of additional features that we have not described in this chapter but that we discuss elsewhere in the book. These are as follows:

- SQL features: various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, case statements, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.
- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases.
- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We called these commands a *storage definition language* (*SDL*). Earlier versions of SQL had commands for **creating indexes**, but these were removed from the language because they were not at the conceptual schema level. Many systems still have the CREATE INDEX commands; but they require a special privilege.
- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.
- SQL has language constructs for specifying the *granting and revoking of privileges* to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or UPDATE—to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called **GRANT** and **REVOKE.**
- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational**. Capabilities such as creating complex-structured attributes, specifying abstract data types (called **UDT**s or user-defined types) for attributes and tables, creating **object identifiers** for referencing tuples, and specifying **operations**.
- SQL and relational databases can interact with new technologies such as XML and OLAP/data warehouses.