MODULE 2
The Relational Data Model and Relational Database Constraints and Relational Algebra

2.1 Relational Model Concepts

- **Domain**: A (usually named) set/universe of *atomic* values, where by "atomic" we mean simply that, from the point of view of the database, each value in the domain is indivisible (i.e., cannot be broken down into component parts).

  Examples of domains (some taken from page 147):

  - USA_phone_number: string of digits of length
  ten - SSN: string of digits of length nine
  - Name: string of characters beginning with an upper case letter
  - GPA: a real number between 0.0 and 4.0
  - Sex: a member of the set { female, male }
  - Dept_Code: a member of the set { CMPS, MATH, ENGL, PHYS, PSYC, ... }

  These are all *logical* descriptions of domains. For implementation purposes, it is necessary to provide descriptions of domains in terms of concrete **data types** (or **formats**) that are provided by the DBMS (such as String, int, boolean), in a manner analogous to how programming languages have intrinsic data types.

- **Attribute**: the *name* of the role played by some value (coming from some domain) in the context of a **relational schema**. The domain of attribute A is denoted dom(A).
- **Tuple**: A tuple is a mapping from attributes to values drawn from the respective domains of those attributes. A tuple is intended to describe some entity (or relationship between entities) in the miniworld.

  As an example, a tuple for a PERSON entity might be

  { Name --> "Rumpelstiltskin", Sex --> Male, IQ --> 143 }

- **Relation**: A (named) set of tuples all of the same form (i.e., having the same set of attributes). The term **table** is a loose synonym. (Some database purists would argue that a table is "only" a physical manifestation of a relation.)
- **Relational Schema**: used for describing (the structure of) a relation. E.g., $R(A_1, A_2, ..., A_n)$ says that R is a relation with *attributes* $A_1, ... A_n$. The **degree** of a relation is the number of attributes it has, here *n*.

  Example: STUDENT(Name, SSN, Address)

(See Figure 5.1, page 149, for an example of a STUDENT relation/table having several tuples/rows.)

One would think that a "complete" relational schema would also specify the domain of each attribute.

- **Relational Database**: A collection of **relations**, each one consistent with its specified relational schema.

2.1.2 Characteristics of Relations

Ordering of Tuples: A relation is a *set* of tuples; hence, there is no order associated with them. That is, it makes no sense to refer to, for example, the 5th tuple in a relation. When a relation is depicted as a table, the tuples are necessarily listed in *some* order, of course, but you should attach no significance to that order. Similarly, when tuples are represented on a storage device, they must be organized in *some* fashion, and it may be advantageous, from a performance standpoint, to organize them in a way that depends upon their content.

Ordering of Attributes: A tuple is best viewed as a mapping from its attributes (i.e., the names we give to the roles played by the values comprising the tuple) to the corresponding values. Hence, the order in which the attributes are listed in a table is irrelevant. (Note that, unfortunately, the set theoretic operations in relational algebra (at least how E&N define them) make implicit use of the order of the attributes. Hence, E&N view attributes as being arranged as a sequence rather than a set.)

Values of Attributes: For a relation to be in *First Normal Form*, each of its attribute domains must consist of atomic (neither composite nor multi-valued) values. Much of the theory underlying the relational model was based upon this assumption. Chapter 10 addresses the issue of including non-atomic values in domains. (Note that in the latest edition of C.J. Date's book, he explicitly argues against this idea, admitting that he has been mistaken in the past.)

The **Null** value: used for *don't know*, *not applicable*.

Interpretation of a Relation: Each relation can be viewed as a **predicate** and each tuple in that relation can be viewed as an assertion for which that predicate is satisfied (i.e., has value **true**) for the combination of values in it. In other words, each tuple represents a fact. Example (see Figure 5.1): The first tuple listed means: There exists a student having name Benjamin Bayer, having SSN 305-61-2435, having age 19, etc.

Keep in mind that some relations represent facts about entities (e.g., students) whereas others represent facts about relationships (between entities). (e.g., students and course sections).

The **closed world assumption** states that the only true facts about the miniworld are those represented by whatever tuples currently populate the database.

2.1.3 Relational Model Notation:

- $R(A_1, A_2, ..., A_n)$ is a relational schema of degree n denoting that there is a relation R having as its attributes $A_1, A_2, ..., A_n$.
- By convention, Q, R, and S denote relation names.
- By convention, q, r, and s denote relation states. For example, $r(R)$ denotes one possible state of relation R. If R is understood from context, this could be written, more simply, as r.
- By convention, t, u, and v denote tuples.
- The "dot notation" $R.A$ (e.g., STUDENT.Name) is used to qualify an attribute name, usually for the purpose of distinguishing it from a same-named attribute in a different relation (e.g., DEPARTMENT.Name).
-

2.2 Relational Model Constraints and Relational Database Schemas

Constraints on databases can be categorized as follows:

- **inherent model-based:** Example: no two tuples in a relation can be duplicates (because a relation is a set of tuples)
- **schema-based:** can be expressed using DDL; this kind is the focus of this section.
- **application-based:** are specific to the "business rules" of the miniworld and typically difficult or impossible to express and enforce within the data model. Hence, it is left to application programs to enforce.

Elaborating upon **schema-based constraints**:

2.2.1 Domain Constraints: Each attribute value must be either **null** (which is really a *non-value*) or drawn from the domain of that attribute. Note that some DBMS's allow you to impose the **not null** constraint upon an attribute, which is to say that that attribute may not have the (non-)value **null**.

2.2.2 Key Constraints: A relation is a *set* of tuples, and each tuple's "identity" is given by the values of its attributes. Hence, it makes no sense for two tuples in a relation to be identical (because then the two tuples are actually one and the same tuple). That is, no two tuples may have the same combination of values in their attributes.

Usually the miniworld dictates that there be (proper) subsets of attributes for which no two tuples may have the same combination of values. Such a set of attributes is called a **superkey** of its relation. From the fact that no two tuples can be identical, it follows that the set of all attributes of a relation constitutes a superkey of that relation.

A **key** is a *minimal superkey*, i.e., a superkey such that, if we were to remove any of its attributes, the resulting set of attributes fails to be a superkey.

Example: Suppose that we stipulate that a faculty member is uniquely identified by *Name* and *Address* and also by *Name* and *Department*, but by no single one of the three attributes mentioned. Then *{ Name, Address, Department }* is a (non-minimal) superkey and each of *{ Name, Address }* and *{ Name, Department }* is a key (i.e., minimal superkey).

Candidate key: any key! (Hence, it is not clear what distinguishes a key from a candidate key.)

Primary key: a key chosen to act as the means by which to identify tuples in a relation.
Typically, one prefers a primary key to be one having as few attributes as possible.

2.2.3 Relational Databases and Relational Database Schemas

A **relational database schema** is a set of schemas for its relations (see Figure 5.5, page 157) together with a set of **integrity constraints**.

A **relational database state/instance/snapshot** is a set of states of its relations such that no integrity constraint is violated. (See Figure 5.6, page 159, for a snapshot of COMPANY.)

2.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

Entity Integrity Constraint: In a tuple, none of the values of the attributes forming the relation's primary key may have the (non-)value **null**. Or is it that at least one such attribute must have a non-null value? In my opinion, E&N do not make it clear!

Referential Integrity Constraint: (See Figure 5.7) A **foreign key** of relation *R* is a set of its attributes intended to be used (by each tuple in *R*) for identifying/referring to a tuple in some relation *S*. (*R* is called the *referencing* relation and *S* the *referenced* relation.) For this to make sense, the set of attributes of *R* forming the foreign key should "correspond to" some superkey of *S*. Indeed, by definition we require this superkey to be the primary key of *S*.

This constraint says that, for every tuple in *R*, the tuple in *S* to which it refers must actually be in *S*. Note that a foreign key may refer to a tuple in the same relation and that a foreign key may be part of a primary key (indeed, for weak entity types, this will always occur). A foreign key may have value **null** (necessarily in all its attributes??), in which case it does not refer to any tuple in the referenced relation.

Semantic Integrity Constraints: application-specific restrictions that are unlikely to be expressible in DDL. Examples:

- salary of a supervisee cannot be greater than that of her/his supervisor
- salary of an employee cannot be lowered

2.3 Update Operations and Dealing with Constraint Violations.

For each of the *update* operations (Insert, Delete, and Update), we consider what kinds of constraint violations may result from applying it and how we might choose to react.

2.3.1 Insert:

- domain constraint violation: some attribute value is not of correct domain
- entity integrity violation: key of new tuple is **null**
- key constraint violation: key of new tuple is same as existing one
- referential integrity violation: foreign key of new tuple refers to non-existent tuple

Ways of dealing with it: reject the attempt to insert! Or give user opportunity to try again with different attribute values.

2.3.2 Delete:

- referential integrity violation: a tuple referring to the deleted one

exists. Three options for dealing with it:

- Reject the deletion
- Attempt to **cascade** (or **propagate**) by deleting any referencing tuples (plus those that reference them, etc., etc.)
- modify the foreign key attribute values in referencing tuples to **null** or to some valid value referencing a different tuple

2.3.3 Update:

- Key constraint violation: primary key is changed so as to become same as another tuple's
- referential integrity violation:
    - foreign key is changed and new one refers to nonexistent tuple
    - primary key is changed and now other tuples that had referred to this one violate the constraint

2.3.4 Transactions: This concept is relevant in the context where multiple users and/or application programs are accessing and updating the database concurrently. A transaction is a logical unit of work that may involve several accesses and/or updates to the database (such as what might be required to reserve several seats on an airplane flight). The point is that, even though several transactions might be processed concurrently, the end result must be as though the transactions were carried out sequentially. (Example of simultaneous withdrawals from same checking account.)

Module 2.4 - SQL

- The name SQL stands for ***Structured Query Language***.
- The SQL language may be considered as one of the major reasons for the success of relational databases in the commercial world.
- SQL is a comprehensive database language because

➢ It has statements for data definition ,database construction and database manipulation
➢ It does automatic query optimizations
➢ It has facilities for defining views on the database
➢ It has facilities for specifying security and authorization
➢ It has facilities for defining integrity constraints
➢ It has facilities for specifying transaction controls
➢ It also has rules for embedding SQL statements into a general-purpose programming language such as Java or COBOL or C/C

4.1 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 SQL command for **data definition** is the **CREATE** statement, which can be used to create **schemas**, **tables** (relations), and **domains** as well as other constructs such as views, assertions, and triggers.

4.1.1 Schema and Catalog Concepts in SQL:

- The database schema concept can be used to ***group together tables and other constructs that belong to the same database application.***
- 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, constraints, views, domains***, and other constructs (such as ***authorization grants***) that describe the schema.
- A schema is created via the **CREATE SCHEMA** statement, which can include all the schema elements' definitions.
- 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 **SMITH**:

  CREATE SCHEMA COMPANY **AUTHORIZATION** SMITH;

- In general, ***not all users are authorized to create schemas and schema elements***. The

privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

- SQL2 uses the concept of a **catalog** - ***a named collection of schemas in an SQL environment.***
- A catalog always contains a special schema called **INFORMATION_SCHEMA**, which provides information on ***all the schemas*** in the catalog and all the element descriptors in these schemas.
- Integrity constraints such as **referential integrity** ***can be defined between relations only if they exist in schemas within the same catalog***.

4.1.2 The CREATE TABLE Command in SQL:

- The **CREATE TABLE** command is used to specify a ***new table*** 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 any ***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, or they can be added later using the **ALTER TABLE** command.
- We can ***explicitly attach the schema name to the relation name***, separated by a period.

  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 relation and its rows are actually created and stored as a file by the DBMS.
- Base relations are distinguished from **"virtual relations"**, created through the **CREATE VIEW** statement, which may or may not correspond to an actual physical file.

- 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. However, rows are not considered to be ordered within a table.
- Figure 8.1 shows sample data definition statements in SQL for the COMPANY database.

```
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),
        SUPERSSN         CHAR(9),
        DNO              INT                NOT NULL,
    PRIMARY KEY (SSN),
    FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE DEPARTMENT
      ( DNAME            VARCHAR(15)        NOT NULL,
        DNUMBER          INT                NOT NULL,
        MGRSSN           CHAR(9)            NOT NULL,
        MGRSTARTDATE     DATE,
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) 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 8.1 SQL CREATE TABLE data definition statements for defining the COMPANY schema

4.1.3 Attribute Data Types and Domains in SQL:

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

a) **Numeric** data types include:

- ***integer numbers*** of various sizes (INTEGER or INT, and SMALLINT).
- ***floating-point (real) numbers*** of various precision (FLOAT or REALand DOUBLE PRECISION).
- ***Formatted numbers*** which 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.

b) **Character-string** data types are either:

- ***fixed length***--CHAR(n) or CHARACTER(n), where *n* is the number of characters.
- ***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.
- For **fixed-length strings**, a ***shorter string is padded with blank characters to the right***.
  For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed.
- ***Padded blanks are generally ignored when strings are compared***. For comparison purposes, strings are considered ordered in alphabetic order;
  If a string **str1** appears before another string ***str2*** in alphabetic order, then *str*1 is considered to be ***less than*** *str*2.
- There is also a **concatenation operator** denoted by || (doublevertical bar) that can concatenate two strings in SQL.
  For example, 'abc' || 'XYZ' results in a single string **'abcXYZ'**.

c) **Bit-string** data types are either of ***fixed length*** -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***.

- ***Literal bit strings*** are placed between ***single quotes*** but preceded by a B to distinguish them from character strings;
  For example, **B'10101**

d) 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***.

e) New data types for **date** and **time** were added in SQL2.

- 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**.

- The **< (less than) comparison** can be used with **dates or times**-an *earlier* date is considered to be smaller than a later date, and similarly with time.

- Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME;
  For example, **DATE '2002-09-27'** or **TIME '09: 12:47'**.

f) A **timestamp** data type (TIMESTAMP) ***includes both the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds.***

- **Literal values** are represented by **single-quoted strings** preceded by the keyword TIMESTAMP, with a blank space between data and time;
  For example, **TIMESTAMP'2002-09-27 09:12:47 648302'**.

g) 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.

Domain

- It is possible to specify the data type of each attribute directly, as in Figure 8.1;
- ***A domain can be declared, and the domain name can be used with the attribute specification***.
  For example, we can create a domain **SSN_TYPE** by the following statement:

  CREATE DOMAIN SSN_TYPE **AS** CHAR(9);

We can use SSN_TYPE in place of CHAR(9) in Figure 8.1 for the attributes SSN and SUPERSSN of EMPLOYEE, MGRSSN of DEPARTMENT, ESSN of WORKS_ON, and ESSN of DEPENDENT.

4.2 SPECIFYING CONSTRAINTS IN SQL:

Basic constraints can be specified in SQL as part of table creation. These include:

- ***key and referential integrity constraints***
- ***restrictions on attribute domains and NULLs***
- ***constraints on individual tuples(rows) within a relation***.

4.2.1 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.

- NOT NULL 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 8.1.

- 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 8.2** illustrates examples of specifying a default values to various attributes.

- ***If no default clause is specified, the default value is NULL for attributes that do not have the NOT NULL constraint***.

```
CREATE TABLE EMPLOYEE
       ( . . . ,
         DNO                INT    NOT NULL    DEFAULT 1,
       CONSTRAINT EMPPK
         PRIMARY KEY (SSN),
       CONSTRAINT EMPSUPERFK
         FOREIGN KEY (SUPERSSN) 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
       ( . . . ,
         MGRSSN    CHAR(9)  NOT NULL DEFAULT '888665555',
         . . . ,
         CONSTRAINT DEPTPK
           PRIMARY KEY (DNUMBER),
         CONSTRAINT DEPTSK
           UNIQUE (DNAME),
         CONSTRAINT DEPTMGRFK
           FOREIGN KEY (MGRSSN) 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 8.2 Example illustrating how default attribute values and referential triggered actions are specified in SQL

- Another type of constraint can restrict attribute or domain values using the **CHECK clause** following an attribute or domain definition.

  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 (see Figure 8.1) 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);

  We can then use the created domain **D_NUM** as the attribute type for all attributes that refer to department numbers in Figure 8.1, such as DNUMBER of DEPARTMENT, DNUM of PROJECT, DNO ofEMPLOYEE, and so on.

4.2.2 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 (secondary) keys***, as illustrated in the DEPARTMENTand PRO] ECT table declarations in Figure 8.1.

- **Referential integrity** is specified via the **FOREIGN KEY** clause

- A referential integrity constraint is violated when rows are inserted or deleted, or when a foreign key or primary key attribute value is modified.

- *The default action that SQL takes for an integrity violation is to reject the update operation that will cause a* violation.

- However, the schema designer can specify an ***alternative action*** to be taken if a referential integrity constraint is violated, by attaching a **referential triggered action** clause to any foreign key constraint.

- The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE as shown in figure 8.2.

- We illustrate this with the examples shown in Figure 8.2. Here, the database designer chooses SET NULL ON DELETE and CASCADE ON UPDATE for the foreign key SUPERSSN of EMPLOYEE (**Figure 8.3)**
  This means that if the row for a supervising employee is *deleted,* the value of SUPERSSN is automatically set to NULL for all employee rows that were referencing the deleted employee tuple. On the other hand, if the SSN value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to SUPERSSN for all employee tuples referencing the updated employee tuple.

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 8.3: One possible database state for the COMPANY database

4.2.3 Giving Names to Constraints:

- Figure 8.2 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.

- A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional.

4.2.4 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 **tuple-based constraints** ***because they apply to each tuple individually and are checked whenever a tuple is inserted or modified***.

  For example, suppose that the DEPARTMENT table in Figure 8.1 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 greater than the department creation date:

  CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);

4.3 SCHEMA CHANGE STATEMENTS IN SQL:

Schema Change commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

4.3.1 The DROP Command:

- The **DROP** command can be used to **drop named schema elements**, such as **tables, domains, or constraints**.

- ***It is also possible to drop a schema***. For example, if a whole schema is not needed any more, the **DROP SCHEMA** command can be used.

- **There are two drop behavior options**: **CASCADE** and **RESTRICT**.

- For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the **CASCADE** option is used as follows:

- **DROP SCHEMA** COMPANY **CASCADE;**

 If the **RESTRICT** option is chosen in place of CASCADE, ***the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed***.

 If a base table within a schema is not needed any longer, the relation and its definition can be deleted by using the **DROP TABLE** command.

For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 8.1, we can get rid of the DEPENDENT relation by issuing the following command:
DROP TABLE DEPENDENT **CASCADE;**

 If the **RESTRICT** option is chosen instead of CASCADE, a ***table is dropped only if it is not referenced in any constraints*** (for example, by foreign key definitions in another relation) ***or views***.

 With the **CASCADE** option, all such ***constraints and views that reference the table are dropped automatically from the schema, along with the table itself***.

 The DROP command can also be used to drop other types of named schema elements, such as ***constraints or domains***.

4.3.2 The ALTER Command:

 The definition of a base table or of other named schema elements can be changed by using the* ALTER *command.

 For base tables, the possible ***alter table actions*** include :

- Adding or dropping a column (attribute)
- Changing a column definition
- Adding or dropping table constraints.

 For example, to **add an attribute for keeping track of jobs of employees** to the **EMPLOYEE** base relations in the COMPANY schema, we can use the command:

ALTER TABLE COMPANY.EMPLOYEE **ADD COLUMN** job **VARCHAR(12)**;

 We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by **Specifying a default clause** or by using the **UPDATE** command

- ***If no default clause is specified, the new attribute will have NULLs*** in all the tuples of the relation immediately after the command is executed;

- To **drop a column**, we must choose either **CASCADE** or **RESTRICT** for drop behavior.
- If **CASCADE** is chosen, ***all constraints and views that reference the column are dropped automatically from the schema, along with the column***.

- If **RESTRICT** is chosen, ***the command is successful only if no views or constraints (or other elements) reference the column***.

- The following command removes the attribute **ADDRESS** from the **EMPLOYEE** base table:
  ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE;**

- It is also possible to alter a column definition by **dropping an existing default clause or by defining a new default clause**.
  The following examples illustrate this clause:

  ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** MGRSSN **DROP DEFAULT;**

  ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** MGRSSN **SET DEFAULT** "333445555";

- ***It is possible to change the constraints specified on a table by adding or dropping a constraint***.

- To be dropped, ***a constraint must have been given a name*** when it was specified.
  For example, to drop the constraint named **EMPSUPERFK** in Figure 8.2 from the EMPLOYEE relation, we write:

- **ALTER TABLE** COMPANY.EMPLOYEE **DROP CONSTRAINT** EMPSUPERFK **CASCADE;**

- ***We can redefine a replacement constraint by adding a new constraint to the relation***, if needed. This is specified by using the **ADD** keyword in the ALTER TABLE statement followed by the new constraint.

4.4 BASIC QUERIES IN SQL:

SQL has one basic statement for ***retrieving information from a database***: the **SELECT** statement.

4.4.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries:

- **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.

- In SQL, the basic ***logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>.***
- These correspond to the relational algebra operators =, <, ~, >, ~, and *, respectively, and to the c{c++ programming language operators =, <, <=, >, >=, and !=.
- **Examples:**

QUERY 0

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

```
Q0: 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 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. **Figure 8.3a** shows the result of query QO.

QUERY 1

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

```
Q1: SELECT FNAME, LNAME, ADDRESS
    FROM   EMPLOYEE, DEPARTMENT
    WHERE  DNAME='Research' AND DNUMBER=DNO;
```

The result of query Ql is shown in **Figure 8.3b**

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 birthdate.

```
Q2: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
    FROM    PROJECT, DEPARTMENT, EMPLOYEE
    WHERE   DNUM=DNUMBER AND MGRSSN=SSN AND
            PLOCATION='Stafford';
```

- ✓ The condition DNUM = DNUMBER relates a project to its controlling department.
- ✓ The condition MGRSSN = SSN relates the controlling department to the employee who manages that department.
- ✓ The condition PLOCATION='Stafford' selects the specified project location.

The result of query Q2 is shown in **Figure 8.3c**.

(a)

BDATE	ADDRESS
1965-01-09	731 Fondren, Houston, TX

(b)

FNAME	LNAME	ADDRESS
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

(c)

PNUMBER	DNUM	LNAME	ADDRESS	BDATE
10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

(d)

E.FNAME	E.LNAME	S.FNAME	S.LNAME
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

(e)

SSN
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

SSN	DNAME
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

(g)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-09-01	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

FIGURE 8.3 Results of SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C

4.4.2 Ambiguous Attribute Names, Aliasing, and Tuple Variables:

- In SQL the ***same name can be used for two (or more) attributes as long as the attributes are in different relations***.

- If this is the case, and ***if a 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 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 Ql would be rephrased as shown in QIA.

```
Q1A: SELECT  FNAME, EMPLOYEE.NAME, ADDRESS
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DEPARTMENT.NAME='Research' AND
             DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

- ***Ambiguity 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.SUPERSSN=S.SSN;
```

- In this case, we are allowed 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, SD, ADDR, SEX, SAL, SSSN, DNO)** in the FROM clause, FN becomes an alias for FNAME, MI for MINH, LN for LNAME, and so on.

- In *Q8,* we can think of **E** and S as two ***different copies*** of the **EMPLOYEE** relation;

➢ The first, **E** represents employees in the role of supervisees

- The second, **S,** represents employees in the role of supervisors. The result of query Q8 is shown in **Figure 8.3d**.

- Whenever one or more aliases are given to a relation, we can use these names to represent different references to that relation. This permits multiple references to the same relation within a query.

- We could specify query **Q1A** as in **Q1B**:

```
Q1B: SELECT  E.FNAME, E.NAME, E.ADDRESS
     FROM    EMPLOYEE E, DEPARTMENT D
     WHERE   D.NAME='Research' AND D.DNUMBER=E.DNUMBER;
```

4.4.3 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 (**Figure 8.3e**), and **Query 10** selects all combinations of an EMPLOYEE SSN and a DEPARTMENT DNAME (**Figure 8.3f**).

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 then incorrect and very large relations may result.

- ***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.*

- Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 8.3g)

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

- Query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works the 'Research' department.

```
Q1D:  SELECT  *
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNAME='Research' AND DNO=DNUMBER;
```

- **Query Ql0A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations**.

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

4.4.4 Tables as Sets in SQL:

- ***An SQL table with a key is restricted to being a set***, since the key value must be distinct in each tuple.

- 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 (SUM,MAX,MIN,AVG) is applied to tuples, in most cases we do not want to eliminate duplicates.

- 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.

- **Query 11 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, as shown in **Figure 8.4(a)**.

- **By using the keyword DISTINCT as in Q11A, we get only the distinct salary values** , as shown in **Figure 8.4(b).**

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;
```

(a)

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

(b)

SALARY
30000
40000
25000
43000
38000
55000

(c)

FNAME	LNAME

(d)

FNAME	LNAME
James	Borg

FIGURE 8.4 Results of additional SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

- **SQL has directly incorporated some of the set operations of relational algebra**.

➢ There is set union (UNION) operation

➢ There is set difference (EXCEPT) operation

➢ And there is set intersection (INTERSECT) operation

- The relations resulting from these set operations are sets of tuples; that is, ***duplicate tuples are eliminated from the result***.

- Because these set operations apply only to ***union-compatible relations***, ***we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations***.

- The following 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.

```
Q4:  (SELECT DISTINCT PNUMBER
      FROM   PROJECT, DEPARTMENT, EMPLOYEE
      WHERE  DNUM=DNUMBER AND MGRSSN=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
- ✓ The second SELECT retrieves the projects that involve a 'Smith' as a worker on the project.
- ✓ Applying the UNION operation to the two SELECT queries gives the desired result.

- **Figure 8.5** illustrates the other multiset operations

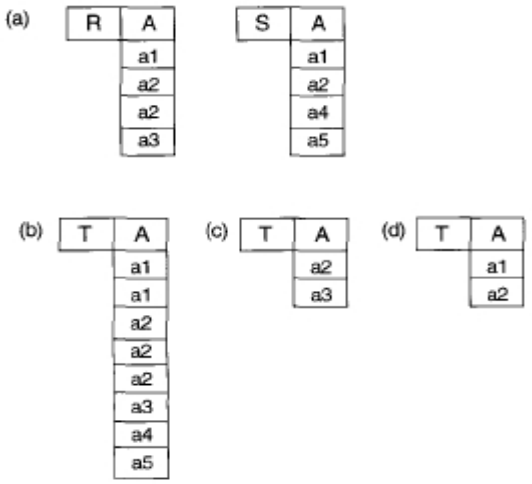


FIGURE 8.5 The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

4.4.5 Substring Pattern Matching and Arithmetic Operators:

- SQL 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.

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 1950s, we can use Query 12A**. Here, '5' 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.

```
Q12A:  SELECT  FNAME, LNAME
       FROM    EMPLOYEE
       WHERE   BDATE LIKE '_ _ 5 _ _ _ _ _ _ _';
```

- ***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.

- 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.***

QUERY 13

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

```
Q13:  SELECT  FNAME, LNAME, 1.1*SALARY AS INCREASED_SAL
      FROM    EMPLOYEE, WORKS_ON, PROJECT
      WHERE   SSN=ESSN AND PNO=PNUMBER AND
              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.

- Another comparison operator that 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)).

4.4.6 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 attributes, using the* ORDER BY *clause.***

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, first name.

```
Q15:  SELECT    DNAME, LNAME, FNAME, PNAME
      FROM      DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
      WHERE     DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER
      ORDER BY  DNAME, LNAME, 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.

4.5. 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.

4.5.1 The INSERT Command:

- In its simplest form, INSERT is used to add a single tuple to a relation. 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, we can use U1:

```
U1: INSERT INTO  EMPLOYEE
    VALUES       ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                 Oak Forest,Katy,TX', 'M', 37000, '987654321', 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 to be 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 UlA are set to their DEFAULT or to NULL.

- 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.

  INSERT INTO EMPLOYEE VALUES (('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest,Katy,TX', 'M', 7000, '987654321', 4) , ('Roy', 'J', 'Mathews', '653298654', '1968-11-23', '94 Fulkerson,NY', 'M', 9000, '987664351', 5));

- A DBMS that fully implements SQL-99 should support and enforce all the integrity constraints that can be specified in the DDL. However, some DBMSs do not incorporate all the constraints (like referential integrity), in order to maintain the efficiency of the DBMS and because of the complexity of enforcing all constraints.

- If a system does not support some constraint, the users or programmers must enforce the constraint.

- For example, if we issue the command in **U2** on the database shown in Table 5.1, a DBMS **not supporting referential integrity** will do the insertion even though no **DEPARTMENT** tuple exists in the database with **DNUMBER = 2**.

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

- It is the responsibility of the user to check that any such constraints *whose checks are not implemented by the* DBMS are not violated.
- A single INSERT command can be used for inserting multiple tuples into a relation in conjunction with creating the relation and loading the relation with the result of a query.

  For example, to create a temporary table DEPT_NAME that has the name, number of employees, and total salaries for each department, we can write the statements in U3A and U3B:

```
U3A: CREATE TABLE  DEPTS_INFO
     (DEPT_NAME    VARCHAR(15),
     NO_OF_EMPS    INTEGER,
     TOTAL_SAL     INTEGER);
U3B: INSERT INTO   DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,
                   TOTAL_SAL)
     SELECT        DNAME, COUNT (*), SUM (SALARY)
     FROM          (DEPARTMENT JOIN EMPLOYEE ON
                   DNUMBER=DNO)
     GROUP BY      DNAME;
```

- We can now query DEPTS_INFO as we would any other relation; when we do not need it any more, we can remove it by using the DROP TABLE command.

4.5.2 The DELETE Command:

- The DELETE command removes tuples from a relation.
- It includes a WHERE clause to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time.
- However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL.
- 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.
- The DELETE commands in U4A to U4D, if applied independently to the database of Table 5.1, 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 IN (SELECT   DNUMBER
                          FROM     DEPARTMENT
                          WHERE    DNAME='Research');
U4D: DELETE FROM  EMPLOYEE;
```

4.5.3 The UPDATE Command:

- The UPDATE command is used to modify attribute values of one or more selected tuples.
- As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation.
- 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.

Example to give all employees in the 'Research' department a 10 percent rise in salary

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

4.6 ADDITIONAL FEATURES OF SQL:

- SQL has the capability to specify more general constraints, called assertions, using the CREATE ASSERTION statement.

- SQL has language constructs for specifying views, also known as virtual tables, using the CREATE VIEW statement. Views are derived from the base tables declared through the CREATE TABLE statement.
- SQL has several different techniques for writing programs in various programming languages that can include SQL statements to access one or more databases. These include embedded SQL, dynamic SQL SQL/CLI (Call Language Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Program Stored Modules).
- 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 call these commands a storage definition language (SDL).
- 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 (also called nested relations), specifying abstract data types (called DDTs or user-defined types) for attributes and tables, creating object identifiers for referencing tuples, and specifying operations on these types.
- SQL and relational databases can interact with new technologies such as XML (eXtended Markup Language) and OLAP (On Line Analytical Processing for Data Warehouses).

Module 2.2-RELATIONAL ALGEBRA

2.1UNARY RELATIONAL OPERATIONS: SELECT and PROJECT

2.1.1The SELECT Operation

- The **SELECT** operation is used to choose a ***subset* of the tuples** from a relation that satisfies a **selection condition**.
- We can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition.
- The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded.

  For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$$\sigma_{Salary>30000}(EMPLOYEE)$$

In general, the SELECT operation is denoted by

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

- 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>

- Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition.

  For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

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

- The 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*. The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative**; that is,

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

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

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ \ldots)) = \sigma_{<cond1>\ \mathbf{AND}<cond2>\ \mathbf{AND}\ldots\mathbf{AND}\ <condn>}(R)$$

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

$$\sigma_{\mathrm{Dno}=4\ \mathbf{AND}\ \mathrm{Salary}>25000}(\mathrm{EMPLOYEE})$$

would correspond to the following SQL query:

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

1.1.2 The PROJECT Operation

- The **PROJECT** operation, 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:

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

- The general form of the PROJECT operation is

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

where (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*.

- The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.
- The **PROJECT** operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. *This is known as* ***duplicate elimination.***

Note:

$$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$$

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:

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

would correspond to the following SQL query:

```
SELECT    DISTINCT Sex, Salary
FROM      EMPLOYEE
```

Consider the relational algebraic queries below:

1.1.3 Sequences of Operations and the RENAME Operation

- We must give names to the relations that hold the intermediate results.
- For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an in-line expression, as follows:

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

- Figure 1.1(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as:

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

:

- It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to rename the attributes in the intermediate and result relations.

(a)

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

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

(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

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

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

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

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

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and B1, B2, ..., Bn are the new attribute names.

- 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,
```

1.2 Relational Algebra Operations from Set Theory

1.2.1 The UNION, INTERSECTION, and MINUS Operations

We can define the ***three set operations UNION, INTERSECTION, and SET DIFFERENCE*** on two union-compatible relations R and S as follows:

- **UNION:** The result of this operation, denoted by R ∪ S, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

  For example, to retrieve the Social Security numbers of all employees who either work in

department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \pi_{Ssn}(DEP5\_EMPS)$$
$$RESULT2(Ssn) \leftarrow \pi_{Super\_ssn}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 6.3), while eliminating any duplicates.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

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

- **INTERSECTION:** The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S.

- **SET DIFFERENCE (or MINUS or EXCEPT):** The result of this operation, denoted by R – S, is a relation that includes all tuples that are in R but not in S.

These are binary operations; that is, each is applied to two sets (of tuples).When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same type of tuples; this condition has been called ***union compatibility or type compatibility***.

Two relations R(A1, A2, ..., An) and S(B1, B2, ..., Bn) are said to be union compatible (or type compatible) if they have the same degree n and if dom(Ai) = dom(Bi) for i 1 to n This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R \quad \text{and} \quad R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as *n*-ary operations applicable to any number of relations because both are also *associative operations;* that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \quad \text{and} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is *not commutative;* that is, in general,

$$R - S \neq S - R$$

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

(a) **STUDENT**

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

INSTRUCTOR

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

(b)

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

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

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

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

1.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

- **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT or CROSS JOIN**—which is denoted by ×.
- This is also a binary set operation, but the relations on which it is applied do not have to be union compatible. This set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set).
- In general, the result of R(A1, A2, ..., An) × S(B1, B2, ..., Bm) is a relation Q with degree n + m attributes Q(A1, A2, ..., An, B1, B2, ..., Bm), in that order.
- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has m tuples and S has n tuples, then R × S will have m*n tuples.

Example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex}='F'}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$
$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

The Cartesian Product (Cross Product) operation.

FEMALE_EMPS

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

EMPNAMES

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

EMP_DEPENDENTS

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

ACTUAL_DEPENDENTS

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

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

1.3 Binary Relational Operations: JOIN and DIVISION

1.3.1 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

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

- The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation.

- Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

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

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

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

The general form of a JOIN operation on two relations[5] $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_m)$ is

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

- The result of the JOIN is a relation Q with $n + m$ attributes $Q(A1, A2, ..., An, B1, B2, ... , Bm)$
  In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result.

DEPT_MGR

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

Figure 6.6
Result of the JOIN operation $DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$.

- 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**.

1.3.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN

- The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. Both previous examples were EQUIJOINs.
- Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple.
- For example, in Figure 6.6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—

denoted by * was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

- The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

- Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.
  In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

  PROJ_DEPT ← 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.

  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

- *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 single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins.**

(a)

PROJ_DEPT

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

(b)

DEPT_LOCS

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

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

- A more general, but nonstandard definition for NATURAL JOIN is

Q ← R *(<list1>),(<list2>)S

In this case, <list1> specifies a list of i attributes from R, and <list2> specifies a list of i attributes from S.

The 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:

$$((\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$

1.3.3 A Complete Set of Relational Algebra Operations

- It has been shown that the set of relational algebra operations {σ, π, ∪, ρ, –, ×} 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 ∩ S ≡ (R ∪ S) – ((R – S) ∪ (S – R))

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

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

A **NATURAL JOIN** can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.

1.3.4 The DIVISION Operation

- The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications.

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:

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

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

$$\text{SSNS(Ssn)} \leftarrow \text{SSN\_PNOS} \div \text{SMITH\_PNOS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname}}(\text{SSNS} * \text{EMPLOYEE})$$

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

(a)

SSN_PNOS

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

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

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

S

A
a1
a2
a3

T

B
b1
b4

- In general, the DIVISION operation is applied to two relations R(Z) ÷ S(X), where the attributes of R are a subset of the attributes of S; that is, X ⊆ Z. Let Y be the set of attributes of R that are not attributes of S;

- The **DIVISION operation** is defined for convenience for dealing with queries that involve ***universal quantification*** or the ***all*** condition

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

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

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{<\text{join condition}>} R_2$, OR $R_1 \bowtie_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$
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 $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

1.3.5 Notation for Query Trees

- Here we describe about the notation typically used in relational systems to represent queries internally. The notation is called a **query tree** or sometimes it is known as a **query evaluation tree** or **query execution tree**.
- It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.
- A **query tree** is a tree data structure that corresponds to a relational algebra expression.
- It represents the *input relations of the query as leaf nodes of the tree*, and represents the *relational algebra operations as internal nodes*.

- An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation.

- The execution terminates when the root node is executed and produces the result relation for the query.

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

Query tree for the abobe query (Q2) . In this, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE.

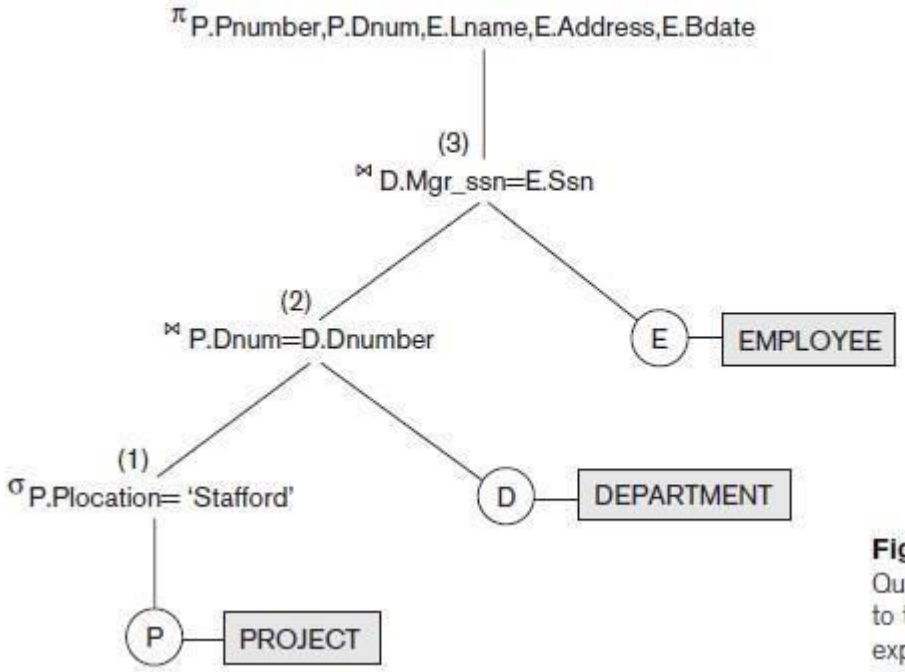


Figure 6.9
Query tree corresponding to the relational algebra expression for Q2.

- In order to execute query Q2, the node marked (1) in Figure 6.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on.

- In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra.

1.4 Additional Relational Operations

1.4.1 Generalized Projection

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

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

where F1, F2, ..., Fn are functions over the attributes in relation R and may involve arithmetic operations and constant values.

Example:

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:

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

1.4.2 Aggregate Functions and Grouping

- Mathematical aggregate functions are applied 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.
- 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 $\mathfrak{I}$ (pronounced script F), to specify these types of requests as follows:

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

  where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM,COUNT—and <attribute> is an attribute of the relation specified by R.
- The resulting relation has the grouping attributes plus one attribute for each element in the function list**.**
- **For example**, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

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

  In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation R.

  If we do not want to rename the attributes then the above query we can write it as,

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

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

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

Example queries and their results:

The aggregate function operation.

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

b. ${}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

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

R

(a)

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

(b)

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

(c)

Count_ssn	Average_salary
8	35125

1.4.3 Recursive Closure Operations

- **Recursive closure** operation in relational algebra is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor.
- This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 3.5 and 3.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor).
- An example of a recursive operation is to retrieve all supervisees of an employee *e* at all levels—that is, all employees *e*' directly supervised by *e*, all employees *e*'' directly supervised by each employee *e*', all employees *e*''' directly supervised by each employee *e*'', and so on.
- 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 3.6), 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:

  RESULT ← RESULT2 ∪ RESULT1

  Example result:

SUPERVISION

(Borg's Ssn is 888665555)

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

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by Borg's subordinates)

RESULT

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 ∪ RESULT2)

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

1.4.4 OUTER JOIN Operations

- For a NATURAL JOIN operation R * S, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a matching (or related) tuple are eliminated from the JOIN result. Tuples with NULL values in the join attributes are also eliminated.
- This type of join, where tuples with no match are eliminated, is known as an inner join. The join operations we described earlier in Section 1.3 are all inner joins.
- This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

- A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values.
- **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:

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

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

- 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 with NULL values.

Figure 6.12
The result of a LEFT OUTER JOIN operation.

RESULT

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

- 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, filling them with NULL values as needed.

1.4.5 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.

- This operation will take the UNION of tuples in two relations R(X, Y) and S(X, Z) that are partially compatible, meaning that only some of their attributes, say X, are union compatible.
- The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation T(X, Y, Z). It is therefore the same as a FULL OUTER JOIN on the common attributes.
- Two tuples t1 in R and t2 in S are said to match if t1[X]=t2[X]. These will be combined (unioned) into a single tuple in t. Tuples in either relation that have no matching tuple in the other relation are padded with NULL values.
- For example, an OUTER UNION can be applied to two relations whose schemas are

  STUDENT(Name, Ssn, Department, Advisor)

  and INSTRUCTOR(Name, Ssn, Department,

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

  STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)
- All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result.
- Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute.
- A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes.

1.5 Examples of Queries in Relational Algebra

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

$$RESEARCH\_DEPT \leftarrow \sigma_{Dname='Research'}(DEPARTMENT)$$
$$RESEARCH\_EMPS \leftarrow (RESEARCH\_DEPT \bowtie_{Dnumber=Dno} EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Fname,\ Lname,\ Address}(RESEARCH\_EMPS)$$

In a single line we can write the above query as,

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

<u>Query 2</u>. Retrieve the number of employees and the average salary of all the employees.

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

Query 3. **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.**

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$$
$$\text{CONTR\_DEPTS} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT})$$
$$\text{PROJ\_DEPT\_MGRS} \leftarrow (\text{CONTR\_DEPTS} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGRS})$$

Query 4. **Find the names of employees who work on all the projects controlled by department number 5.**

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

Query 5. **Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.**

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

Query 6. **List the names of all employees with two or more dependents.**

$$T1(\text{Ssn, No\_of\_dependents}) \leftarrow {}_{\text{Essn}}\Im_{\text{COUNT Dependent\_name}}(\text{DEPENDENT})$$
$$T2 \leftarrow \sigma_{\text{No\_of\_dependents}>2}(T1)$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(T2 * \text{EMPLOYEE})$$

Query 7. **Retrieve the names of employees who have no dependents.**

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

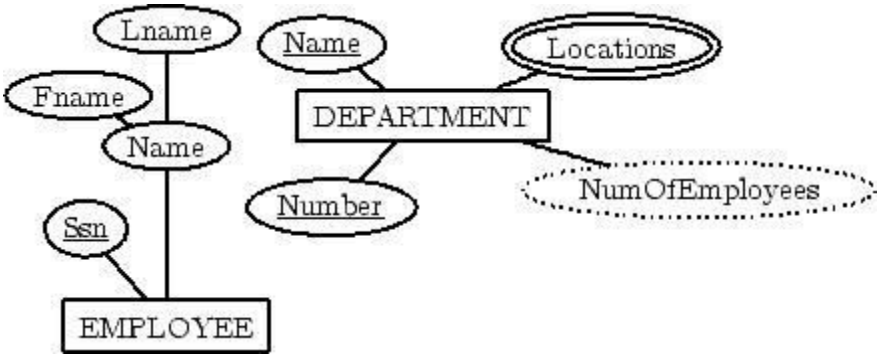
Query 8. **List the names of managers who have at least one dependent.**

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

Module 2.3 Mapping conceptual Design into a logical design

Relational Database Design Using ER-to-Relational Mapping

Step 1: For each **regular (strong) entity type** E in the ER schema, create a relation R that includes all the simple attributes of E.



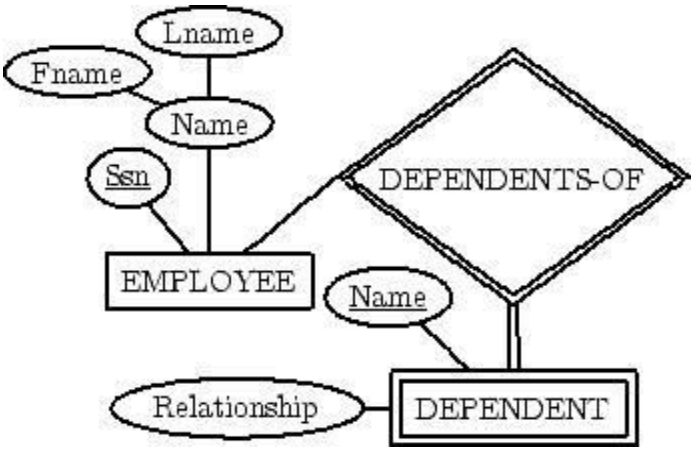
EMPLOYEE

SSN	Lname	Fname

DEPARTMENT

NUMBER	NAME

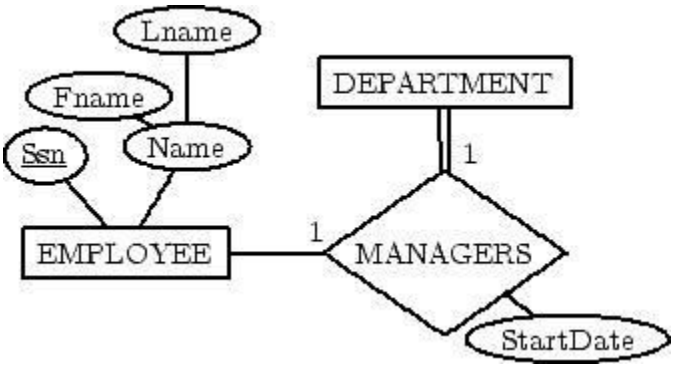
Step 2: For each **weak entity type** W in the ER schema with owner entity type E, create a relation R, and include all simple attributes (or simple components of composite attributes) of W as attributes. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).



DEPENDENT

EMPL-SSN	NAME	Relationship

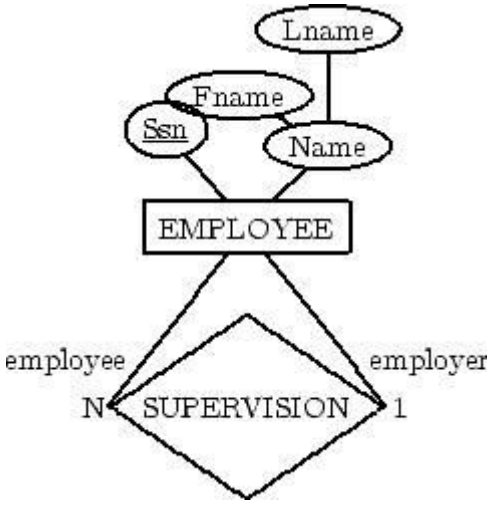
Step 3: For each **binary 1:1 relationship type** R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. Choose one of the relations, say S, and include the primary key of T as a foreign key in S. Include all the simple attributes of R as attributes of S.



DEPARTMENT

MANAGER-SSN	StartDate

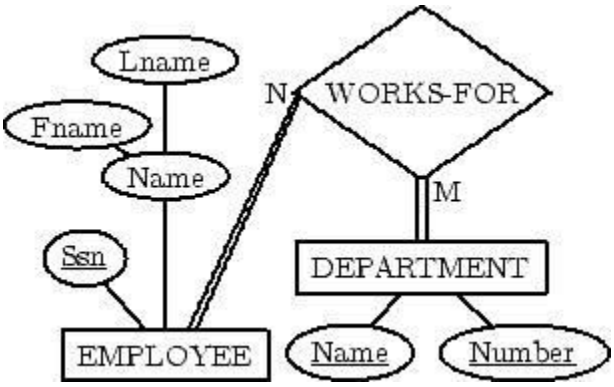
Step 4: For each regular **binary 1:N relationship type** R identify the relation (N) relation S. Include the primary key of T as a foreign key of S. Simple attributes of R map to attributes of S.



EMPLOYEE

SupervisorSSN

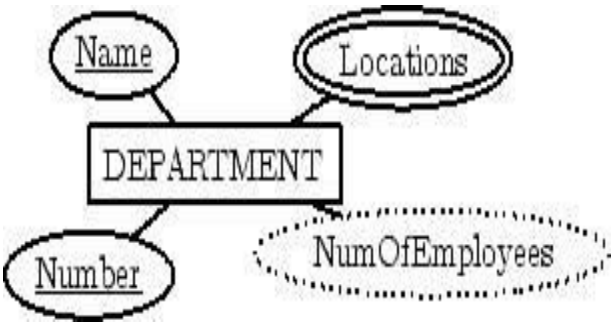
Step 5: For each **binary M:N relationship type** R, create a relation S. Include the primary keys of participant relations as foreign keys in S. Their combination will be the primary key for S. Simple attributes of R become attributes of S.



WORKS-FOR

EmployeeSSN	DeptNumber

Step 6: For each **multi-valued attribute A**, create a new relation R. This relation will include an attribute corresponding to A, plus the primary key K of the parent relation (entity type or relationship type) as a foreign key in R. The primary key of R is the combination of A and K.



DEP-LOCATION

Location	DEP-NUMBER

Step 7: For each **n-ary relationship type R**, where n>2, create a new relation S to represent R. Include the primary keys of the relations participating in R as foreign keys in S. Simple attributes of R map to attributes of S. The primary key of S is a combination of all the foreign keys that reference the participants that have cardinality constraint > 1.

For a recursive relationship, we will need a new relation.

Questions

1. Define the following terms with an example for each.
2. Explain:
3. i) Domain constraint ii) Semantic integrity constraint iii) Functional dependency constraint
4. List the characteristics of relation? Discuss any one?
5. Discuss various types of Inner Join Operations?
6. Discuss the characteristics of a relation, with an example
7. Briefly discuss the different types of update operations on relational database. show an example of
8. What is valid state and an invalid state,with respect to a database
9. Define referential integrity constraint. Explain the importance of referential integrity constraint. How is this constraint implemented in SQL
10. Define referential integrity in each of the update operation