

## UNIT-2

### SYLLABUS:

**Relational Model:** The Relational Model Concepts, Relational Model Constraints and Relational Database Schemas.

**SQL:** Data Definition, Constraints, and Basic Queries and Updates, SQL Advanced Queries, Assertions, Triggers, and Views

**Formal Relational Languages:** Relational Algebra: Unary Relational Operations: Select and Project, Relational Algebra Operations from Set Theory, Binary Relational Operations: Join and Division, Examples of Queries in Relational Algebra.

### **1. Relational Model:**

#### **Introduction**

The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd 1970), and it attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a mathematical relation which looks somewhat like a table of values as its basic building block, and has its theoretical basis in set theory and first-order predicate logic.

The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems. Current popular relational DBMSs (RDBMSs) include DB2 and Informix Dynamic Server (from IBM), Oracle and Rdb (from Oracle), Sybase DBMS (from Sybase) and SQLServer and Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

#### **1.1 The Relational Model Concepts:**

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent, a flat file of records. It is called a flat file because each record has a simple linear or flat structure.

When a relation is thought of as a table of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

**Example:** In STUDENT relation because each row represents facts about a particular student entity. The column names Name, Student\_number, Class, and Major specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a tuple, a column header is called an attribute, and the table is called a relation. The data type describing the types of values that can appear in each column is represented by a domain of possible values.

##### **1.1.1 Domains, Attributes, Tuples, and Relations**

#### **Domain:**

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is invisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify the name for the domain, to help in interpreting its values.

Some examples of domains follow:

- Usa\_phone\_numbers: The set of ten-digit phone numbers valid in United States.
- Social\_security\_numbers: The set of valid nine-digit social security numbers.
- Names: The set of character strings that represents the names of persons.
- Employee\_ages: Possible ages of employees in a company; each must be an integer value between 15 and 80.

The preceding are called logical definitions of domains. A data type or format is also specified for each domain. For example, the data type for the domain Usa\_phone\_numbers can be declared as a character string of the form (ddd)dddddd, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee\_ages is an integer number between 15 and 80.

**Attribute:**

An **attribute**  $A_i$  is the name of a role played by some domain D in the relation schema R. D is called the domain of  $A_i$  and is denoted by  $dom(A_i)$ .

**Tuple:**

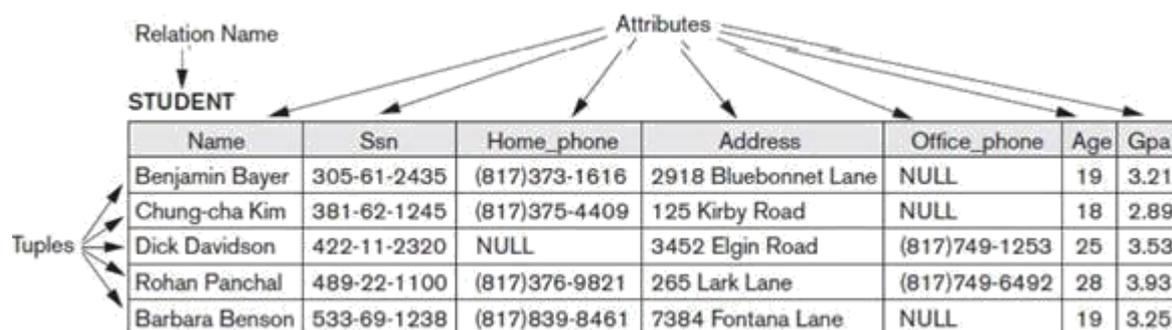
Mapping from attributes to values drawn from the respective domains of those attributes. Tuples are intended to describe some entity (or relationship between entities) in the miniworld

**Example:** a tuple for a PERSON entity might be

{ Name --> "smith", Gender --> Male, Age --> 25 }

**Relation:**

A named set of tuples all of the same form i.e., having the same set of attributes.



**Relation schema:**

A relation schema R, denoted by  $R(A_1, A_2, \dots, A_n)$ , is made up of a relation name R and a list of attributes  $A_1, A_2, \dots, A_n$ . Each attribute  $A_i$  is the name of a role played by some domain D in the relation schema R. D is called the domain of  $A_i$  and is denoted by  $dom(A_i)$ . A relation schema is used to describe a relation; R is called the name of this relation.

The degree (or arity) of a relation is the number of attributes  $n$  of its relation schema. A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

**STUDENT(Name, Ssn, Home\_phone, Address, Office\_phone, Age, Gpa)**

Using the data type of each attribute, the definition is sometimes written as:

**STUDENT(Name: string, Ssn: string, Home\_phone: string, Address: string, Office\_phone: string, Age: integer, Gpa: real)**

Domains for some of the attributes of the STUDENT relation:

dom(Name) = Names;

dom(Ssn) = Social\_security\_numbers;

dom(HomePhone) =USA\_phone\_numbers,

dom(Office\_phone) = USA\_phone\_numbers,

**Relation (or relation state):**

A relation (or relation state)  $r$  of the relation schema by  $R(A_1, A_2, \dots, A_n)$ , also denoted by  $r(R)$ , is a set of  $n$ -tuples  $r = \{t_1, t_2, \dots, t_m\}$ . Each  $n$ -tuple  $t$  is an ordered list of  $n$  values  $t = \langle v_1, v_2, \dots, v_n \rangle$  where each value  $v_i \leq i \leq n$  is an element of  $\text{dom}(A_i)$  or is a special NULL value. The  $i^{\text{th}}$  value in tuple  $t$ , which corresponds to the attribute  $A_i$ , is referred to as  $t[A_i]$  or  $t.A_i$ .

The terms relation intension for the schema  $R$  and relation extension for a relation state  $r(R)$  are also commonly used.

| <u>Informal Terms</u>      | <u>Formal Terms</u>   |
|----------------------------|-----------------------|
| Table                      | Relation              |
| Column Header              | Attribute             |
| All possible Column Values | Domain                |
| Row                        | Tuple                 |
| Table Definition           | Schema of a Relation  |
| Populated Table            | State of the Relation |

**1.1.2 Characteristics of Relations**

**1) Ordering of Tuples in a Relation**

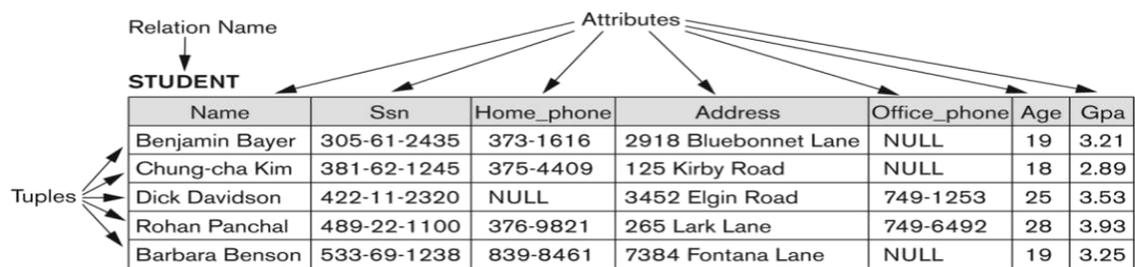
A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them; hence, tuples in a relation do not have any particular order. Tuple ordering

is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation.

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

The order of attributes and their values is not that important as long as the correspondence between attributes and values is maintained. An alternative definition of a relation can be given, making the ordering of values in a tuple unnecessary. In this definition A relation schema  $R(A_1, A_2, \dots, A_n)$ , set of attributes and a relation state  $r(R)$  is a finite set of mappings  $r = \{t_1, t_2, \dots, t_m\}$ , where each tuple  $t_i$  is a mapping from  $R$  to  $D$ .

According to this definition of tuple as a mapping, a tuple can be considered as a set of ( $\langle$ attribute $\rangle$ ,  $\langle$ value $\rangle$ ) pairs, where each pair gives the value of the mapping from an attribute  $A_i$  to a value  $v_i$  from  $dom(A_i)$ . The ordering of attributes is not important, because the attribute name appears with its value.



**Figure 5.1**

The attributes and tuples of a relation STUDENT.

**Figure 5.2**

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

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

**3) Values and NULLs in the Tuples**

Each value in a tuple is atomic. NULL values are used to represent the values of attributes that may be unknown or may not apply to a tuple. For example some STUDENT tuples have NULL for their office phones because they do not have an office. Another student has a NULL for home phone. In general, we can have several meanings for NULL values, such as value unknown, value exists but is not available, or attribute does not apply to this tuple (also known as value undefined).

**4) Interpretation (Meaning) of a Relation**

The relation schema can be interpreted as a declaration or a type of assertion. For example, the schema of the STUDENT relation asserts that, in general, a student entity has a Name, Ssn, Home\_phone, Address, Office\_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a particular instance of the assertion. For example, the first tuple asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

An alternative interpretation of a relation schema is as a predicate; in this case, the values in each tuple are interpreted as values that satisfy the predicate.

### 1.1.3 Relational Model Notation

- Relation schema  $R$  of degree  $n$  is denoted by  $R(A_1, A_2, \dots, A_n)$
- Uppercase letters  $Q, R, S$  denote relation names
- Lowercase letters  $q, r, s$  denote relation states
- Letters  $t, u, v$  denote tuples
- In general, the name of a relation schema such as **STUDENT** also indicates the current set of tuples in that relation
- An attribute  $A$  can be qualified with the relation name  $R$  to which it belongs by using the dot notation  $R.A$  for example, **STUDENT.Name** or **STUDENT.Age**
- An  $n$ -tuple  $t$  in a relation  $r(R)$  is denoted by  $t = \langle v_1, v_2, \dots, v_n \rangle$ , where  $v_i$  is the value corresponding to attribute  $A_i$ . The following notation refers to component values of tuples: Both  $t[A_i]$  and  $t.A_i$  (and sometimes  $t[i]$ ) refer to the value  $v_i$  in  $t$  for attribute  $A_i$ .
- Both  $t[A_u, A_w, \dots, A_z]$  and  $t.(A_u, A_w, \dots, A_z)$ , where  $A_u, A_w, \dots, A_z$  is a list of attributes from  $R$ , refer to the subtuple of values  $\langle v_u, v_w, \dots, v_z \rangle$  from  $t$  corresponding to the attributes specified in the list.

## 1.2 Relational Model Constraints and Relational Database Schemas:

**Constraints** are restrictions on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents. Constraints on databases can generally be divided into three main categories:

### 1) **Inherent model-based constraints or implicit constraints**

- Constraints that are inherent in the data model.
- The characteristics of relations are the inherent constraints of the relational model and belong to the first category. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint.

### 2) **Schema-based constraints or explicit constraints**

- Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL.
- The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

### 3) **Application-based or semantic constraints or business rules**

- Constraints that cannot be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs.
- Examples of such constraints are the salary of an employee should not exceed the salary of the employee, supervisor and the maximum number of hours an employee can work on all projects per week is 56.

#### 1.2.1 Domain Constraints

Domain Constraints specify that within each tuple, the value of each attribute  $A$  must be an atomic value from the domain  $\text{dom}(A)$ . The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and doubleprecision float). Characters, Booleans, fixed-length

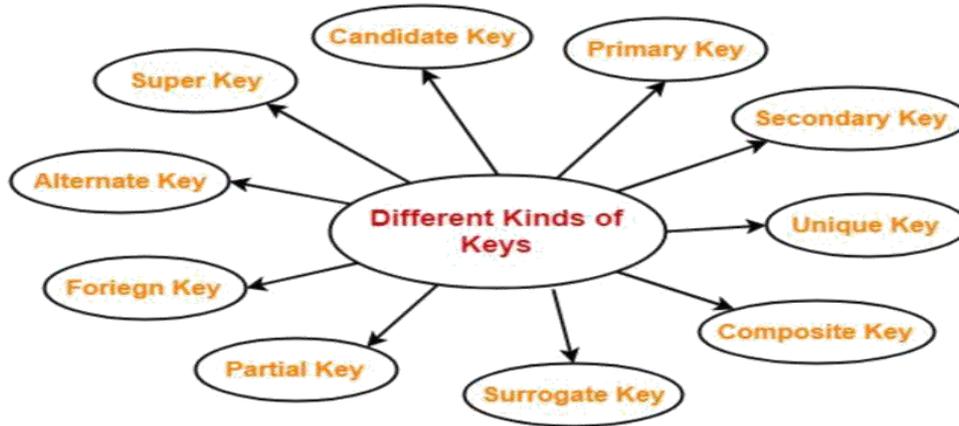
strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types.

**1.2.2 Key Constraints and Constraints on NULL Values**

A **key** is a set of one or more attributes that can uniquely identify each row in a table. A key not only identifies the rows of a table but also relates two or more tables.

Different Types of Keys:

- 1) Super Key
- 2) Candidate Key
- 3) Primary Key
- 4) Foreign Key
- 5) Secondary Key/Alternate Key
- 6) Unique Key
- 7) Composite Key
- 8) Surrogate Key
- 9) Partial Key



- 1) **Super Key:** Super Key is an attribute (or a set of attributes) that uniquely identify a tuple i.e. an entity in entity set.

It is a superset of Candidate Key, since Candidate Keys are selected from super key.

**Example:**

<Student>

| Student_ID | Student_Enroll | Student_Name | Student_Email |
|------------|----------------|--------------|---------------|
| S02        | 4545           | Dave         | ddd@gmail.com |
| S34        | 4541           | Jack         | jjj@gmail.com |
| S22        | 4555           | Mark         | mmm@gmail.com |

Super Keys are:

- {Student\_ID}
- {Student\_Enroll}
- {Student\_Email}
- {Student\_ID, Student\_Enroll}
- {Student\_ID, Student\_Name}
- {Student\_ID, Student\_Email}
- {Student\_Name, Student\_Enroll}
- {Student\_ID, Student\_Enroll, Student\_Name}
- {Student\_ID, Student\_Enroll, Student\_Email}
- {Student\_ID, Student\_Enroll, Student\_Name, Student\_Email}

Candidate Keys are:

- {Student\_ID}
- {Student\_Enroll}
- {Student\_Email}

- 2) **Candidate Key:** Each table has only a single primary key. Each relation may have one or more candidate key. One of these candidate key is called Primary Key. Each candidate key qualifies for Primary Key. Therefore candidates for Primary Key is called Candidate Key.

Candidate key can be a single column or combination of more than one column. A minimal super key is called a candidate key.

**Example:**

| Student_ID | Student_Enroll | Student_Name | Student_Email |
|------------|----------------|--------------|---------------|
| S02        | 4545           | Dave         | ddd@gmail.com |
| S34        | 4541           | Jack         | jjj@gmail.com |
| S22        | 4555           | Mark         | mmm@gmail.com |

Above, Student\_ID, Student\_Enroll and Student\_Email are the candidate keys. They are considered candidate keys since they can uniquely identify the student record.

- 3) **Primary Key:** It is an attribute or set of attributes that uniquely identify an entity (row) in the entity set (table). The main difference between the primary key and the candidate key in that is primary key does not contain NULL values.

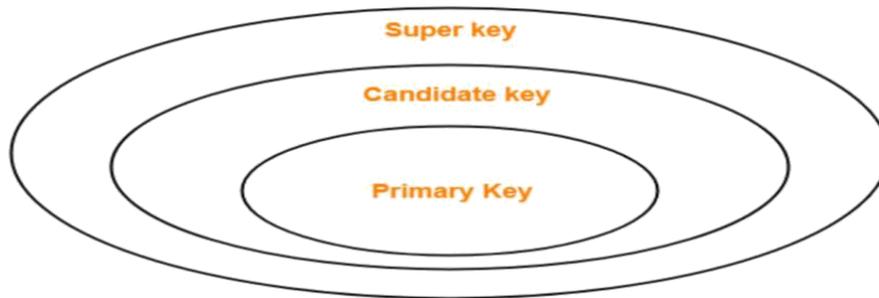
✓ Primary Key must be UNIQUE and NOT NULL.

**Example:**

**EMPLOYEE**

| EID | EMP_FNAME  | EMP_LNAME  |
|-----|------------|------------|
| 12  | Sai        | Keerthi    |
| 13  | Potluri    | Siddhartha |
| 14  | Velagapudi | Krishna    |
| 15  | Rallapalli | Suma       |

The primary key of the relation can be EID.



4) **Foreign Key:** A foreign key is a set of attributes in a table that refers to the primary key of another table. The foreign key links these two tables.

**Example:**

| STUDENT_STATUS       |                 |              | STUDENT_DETAILS |                      |                 |                   |              |               |
|----------------------|-----------------|--------------|-----------------|----------------------|-----------------|-------------------|--------------|---------------|
| <b>Primary Key</b> ↑ | <b>SROLL_NO</b> | <b>SNAME</b> | <b>DOB</b>      | <b>Foreign Key</b> ↑ | <b>SROLL_NO</b> | <b>PROJECT_ID</b> | <b>MARKS</b> | <b>STATUS</b> |
|                      | 12              | Keerthi      | 10-03-1999      |                      | 12              | P1                | 100          | Pass          |
|                      | 13              | Siddhartha   | 18-06-2001      |                      | 13              | P2                | 90           | Pass          |
|                      | 14              | Krishna      | 21-09-2000      |                      | 14              | P3                | 80           | Pass          |
|                      | 15              | Suma         | 30-01-1998      |                      |                 |                   |              |               |

5) **Secondary Key/Alternate Key:** A primary key is the field in a database that is the primary key used to uniquely identify a record in a database. A secondary key is an additional key, or alternate key, which can be use in addition to the primary key to locate specific data.

Secondary Key is the key that has not been selected to be the primary key. However, it is considered a candidate key for the primary key.

Therefore, a candidate key not selected as a primary key is called secondary key. Candidate key is an attribute or set of attributes that you can consider as a Primary key. Note: Secondary Key is not a Foreign Key.

**Example 1:**

| Student_ID | Student_Enroll | Student_Name | Student_Age | Student_Email |
|------------|----------------|--------------|-------------|---------------|
| 096        | 9122717        | Manish       | 25          | aaa@gmail.com |
| 055        | 9122655        | Manan        | 23          | abc@gmail.com |
| 067        | 9122699        | Shreyas      | 28          | pqr@gmail.com |

Above, Student\_ID, Student\_Enroll and Student\_Email are the candidate keys. They are considered candidate keys since they can uniquely identify the student record. Select any one of the candidate key as the primary key. Rest of the two keys would be Secondary Key.

If you selected Student\_ID as primary key, therefore Student\_Enroll and Student\_Email will be Secondary Key (candidates of primary key).

**Example 2:**

| Employee_ID | Employee_No | Employee_Name | Employee_Email    | Employee_Dept |
|-------------|-------------|---------------|-------------------|---------------|
| 0989        | E7897       | Jacob         | jacob@example.com | Finance       |
| 0777        | E8768       | Anna          | anna@example.com  | HR            |
| 0656        | E8789       | Tom           | tom@example.com   | Operations    |

Above, Employee\_ID, Employee\_No and Employee\_Email are the candidate keys. They uniquely identify the Employee record. Select any one of the candidate key as the primary key. Rest of the two keys would be Secondary Key.

- 6) **Unique Key:** A Unique Key is used to prevent duplicate values in a column. Primary Key provided uniqueness to a table.

A primary key cannot accept NULL values; this makes Primary Key different from Unique Key, since Unique Key allows one value as NULL value.

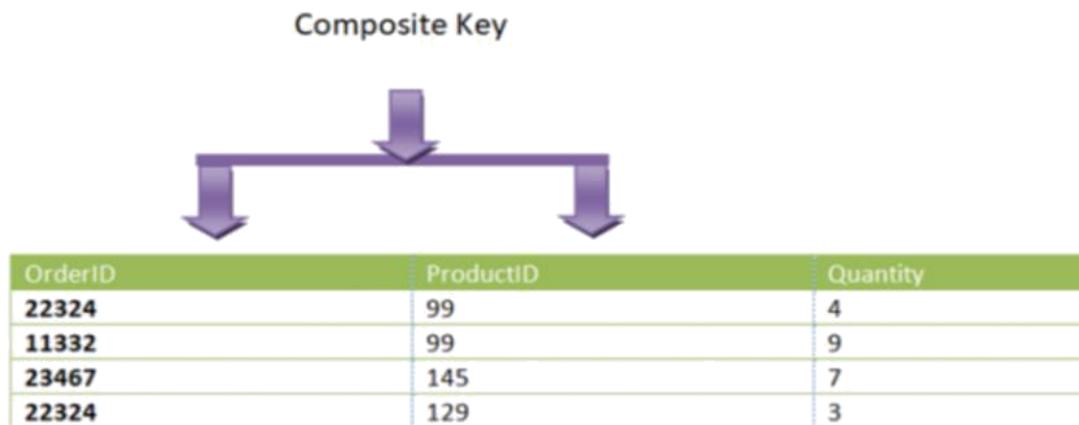
A table can only have a single Primary Key, whereas a Unique Key can be more than one if you need it in the table.

Unique Key ensures that data is not duplicated in two rows in the database. A row in the database can have null in case of Unique Key.

You cannot modify a Primary Key, but a Unique Key can be modified.

- 7) **Composite Key:** A primary key having two or more attributes is called composite key. It is a combination of two or more columns.

**Example 1:** Here our composite key is OrderID and ProductID – {OrderID, ProductID}



**Example 2:**  
<Student>

| StudentID | StudentEnrollNo | StudentMarks | StudentPercentage |
|-----------|-----------------|--------------|-------------------|
| S001      | 0721722         | 570          | 90                |
| S002      | 0721790         | 490          | 80                |
| S003      | 0721766         | 440          | 86                |

Above, our composite keys are StudentID and StudentEnrollNo. The table has two attributes as primary key.

Therefore, the Primary Key consisting of two or more attribute is called Composite Key.

- 8) **Surrogate Key:** A Surrogate Key’s only purpose is to be a unique identifier in a database, for example, incremental key.

Surrogate Key has no actual meaning and is used to represent existence. It has an existence only for data analysis.

**Example:** The surrogate key is Key in the <ProductPrice> table.

<ProductPrice>

| Key    | ProductID | Price |
|--------|-----------|-------|
| 505_92 | 1987      | 200   |
| 698_56 | 1256      | 170   |
| 304_57 | 1898      | 250   |
| 458_66 | 1666      | 110   |

**Other examples of a Surrogate Key:**

- ✓ Counter
- ✓ System date/time stamp
- ✓ Random alphanumeric string.

9) **Partial Key:** Partial key is a key using which all the records of the table can not be identified uniquely.

However, a bunch of related tuples can be selected from the table using the partial key. **Example:** Consider the following schema-

Department ( Emp\_no , Dependent\_name , Relation )

| Emp_no | Dependent_name | Relation |
|--------|----------------|----------|
| E1     | Suman          | Mother   |
| E1     | Ajay           | Father   |
| E2     | Vijay          | Father   |
| E2     | Ankush         | Son      |

Here, using partial key Emp\_no, we can not identify a tuple uniquely but we can select a bunch of tuples from the table

Following are the important differences between Primary Key and Candidate key.

| Sr. No. | Key        | Primary Key  | Candidate key   |
|---------|------------|--|---|
| 1       | Definition | Primary Key is a unique and non-null key which identify a record uniquely in table. A table can have only one primary key. | Candidate key is also a unique key to identify a record uniquely in a table but a table can have multiple candidate keys. |
| 2       | Null       | Primary key column value can not be null.  | Candidate key column can have null value.   |
| 3       | Objective  | Primary key is most important part of any relation or table.   | Candidate key signifies as which key can be used as Primary Key.  |
| 4       | Use        | Primary Key is a candidate key.  | Candidate key may or may not be a primary key.  |

Following are the important differences between Super Key and Candidate key.

| Sr. No. | Key        | Super Key  | Candidate key   |
|---------|------------|--|---|
| 1       | Definition | Super Key is used to identify all the records in a relation. | Candidate key is a subset of Super Key.                       |
| 2       | Use        | All super keys can't be candidate keys.                      | All candidate keys are super keys.                            |
| 3       | Selection  | Super keys are combined together to create a candidate key.  | Candidate keys are combined together to create a primary key. |
| 4       | Count Wise | Super keys are more than Candidate keys.                     | Candidate keys are less than Super Keys.                      |

**Primary Key**

- It is used to ensure that the data in the specific column is unique.
- It helps uniquely identify a record in a relational database.
- One primary key only is allowed in a table.
- It is a combination of the 'UNIQUE' and 'Not Null' constraints.
- This means it can't be a NULL value.
- It is the most important part of a table.
- It is a candidate key.
- Its value can't be deleted from parent table.
- The constraint can be implicitly defined for the temporary tables.

**Candidate key**

- It can have NULL value.
- It may or may not have a primary key.
- It tells about which key can be used as a primary key.
- It is a unique key that helps identify a record uniquely in a table.
- A table can have multiple candidate keys.

All tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. There are other subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes.

Suppose that we denote one such subset of attributes by SK; then for any two distinct tuples t1 and t2 in a relation state r of R, we have the constraint that:  $t_1[SK] \neq t_2[SK]$  . such set of attributes SK is called a superkey of the relation schema R

**Superkey**

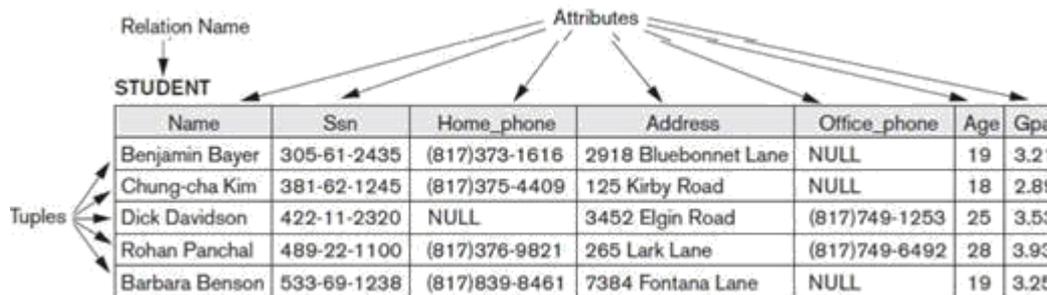
A superkey SK specifies a uniqueness constraint that no two distinct tuples in any state r of R can have the same value for SK. Every relation has at least one default superkey the set of all its attributes.

**Key**

A key K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R anymore. Hence, a key satisfies two properties:

- Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.
- It is a minimal superkey that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition will hold. This property is not required by a superkey.

**Example:** Consider the STUDENT relation



- The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.
- Any set of attributes that includes Ssn for example, {Ssn, Name, Age} is a superkey.
- The superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey.

In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require all its attributes together to have the uniqueness property.

**Candidate Key**

A relation schema may have more than one key. In this case, each of the keys is called a candidate key.

**Example:** The CAR relation has two candidate keys: License\_number and Engine\_serial\_number

CAR

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

**Primary Key**

It is common to designate one of the candidate keys as the primary key of the relation. This is the candidate key whose values are used to identify tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined. Other candidate keys are designated as unique keys and are not underlined.

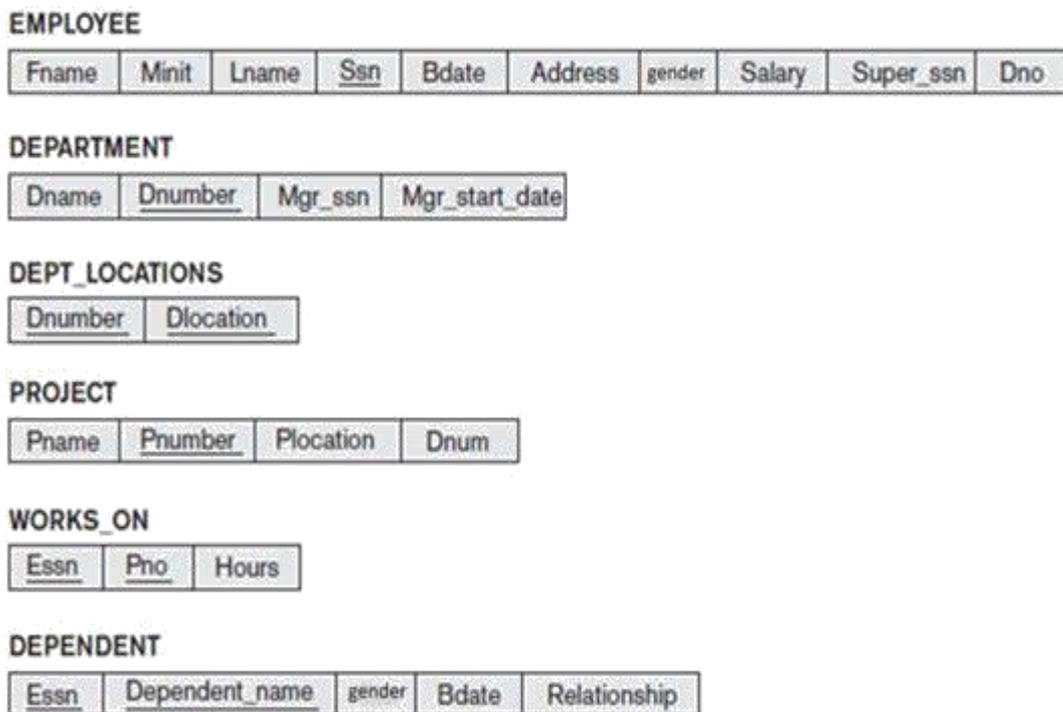
Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

**1.2.3 Relational Databases and Relational Database Schemas**

Relational database schema S is a set of relation schemas  $S = \{R_1, R_2, \dots, R_m\}$  and a set of integrity constraints IC.

Example of relational database schema:

**COMPANY = {EMPLOYEE, DEPARTMENT, DEPT\_LOCATIONS, PROJECT, WORKS\_ON, DEPENDENT}**



*Figure: Schema diagram for the COMPANY relational database schema. The underlined attributes represent primary keys*

A Relational database state is a set of relation states  $DB = \{r_1, r_2, \dots, r_m\}$ . Each  $r_i$  is a state of  $R_i$  and such that the  $r_i$  relation states satisfy integrity constraints specified in IC.

EMPLOYEE

| Empno    | Minit | Lname   | Emp       | Bdate      | Address                  | gender | Salary | Super_emp | Dno |
|----------|-------|---------|-----------|------------|--------------------------|--------|--------|-----------|-----|
| John     | B     | Smith   | 123456789 | 1985-01-09 | 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     | Zelays  | 999887777 | 1968-01-19 | 3321 Castle, Spring, TX  | F      | 25000  | 987654321 | 4   |
| Jennifer | S     | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellair, TX   | F      | 43000  | 888665555 | 4   |
| Ramesh   | K     | Narayan | 888884444 | 1962-09-15 | 970 Fire Oak, Humble, TX | M      | 38000  | 333445555 | 5   |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5431 Rice, Houston, TX   | F      | 25000  | 333445555 | 5   |
| Abner    | 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_emp   | 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

| Empno     | Pno | Hours |
|-----------|-----|-------|
| 123456789 | 1   | 32.5  |
| 123456789 | 2   | 7.5   |
| 888884444 | 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

| Empno     | Dependent_name | gender | Bdate      | Relationship |
|-----------|----------------|--------|------------|--------------|
| 333445555 | Alice          | F      | 1986-04-05 | Daughter     |
| 333445555 | Theodore       | M      | 1983-10-25 | Son          |
| 333445555 | Joy            | F      | 1958-05-09 | Spouse       |
| 987654321 | Abner          | M      | 1943-03-28 | Spouse       |
| 123456789 | Michael        | M      | 1988-01-04 | Son          |
| 123456789 | Alice          | F      | 1988-12-30 | Daughter     |
| 123456789 | Elizabeth      | F      | 1967-05-05 | Spouse       |

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

A database state that does not obey all the integrity constraints is called Invalid state and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a Valid state.

Attributes that represent the same real-world concept may or may not have identical names in different relations.

**Example:** The Dnumber attribute in both DEPARTMENT and DEPT\_LOCATIONS stands for the same real-world concept the number given to a department.

That same concept is called Dno in EMPLOYEE and Dnum in PROJECT.

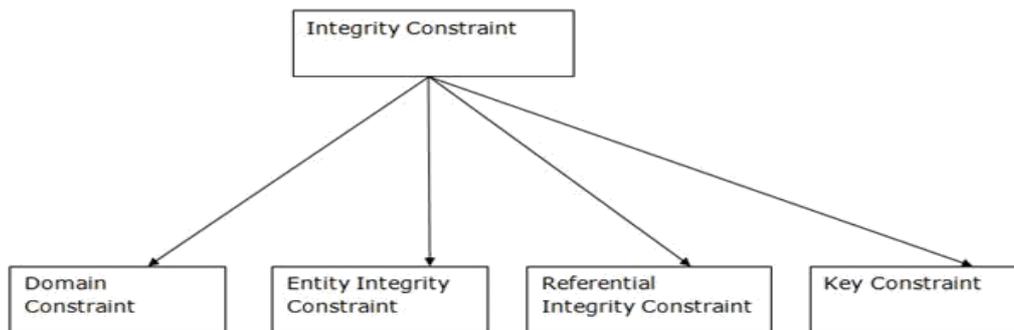
Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different real world concepts project names and department names.

**1.2.4 Integrity, Referential Integrity, and Foreign Keys**

**Integrity Constraints**

- ✓ Integrity constraints are a set of rules. It is used to maintain the quality of information.
- ✓ Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- ✓ Thus, integrity constraint is used to guard against accidental damage to the database. Types of Integrity Constraint

**Types of Integrity Constraint**



**1) Domain Constraints:**

- ✓ Domain constraints can be defined as the definition of a valid set of values for an attribute.
- ✓ The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

**Example:**

| ID   | NAME     | SEMENSTER       | AGE |
|------|----------|-----------------|-----|
| 1000 | Tom      | 1 <sup>st</sup> | 17  |
| 1001 | Johnson  | 2 <sup>nd</sup> | 24  |
| 1002 | Leonardo | 5 <sup>th</sup> | 21  |
| 1003 | Kate     | 3 <sup>rd</sup> | 19  |
| 1004 | Morgan   | 8 <sup>th</sup> | A   |

Not allowed. Because AGE is an integer attribute

**2) Entity integrity constraints**

- ✓ The entity integrity constraint states that primary key value can't be null.
- ✓ This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- ✓ A table can contain a null value other than the primary key field.

**Example:**

**EMPLOYEE**

| EMP_ID | EMP_NAME | SALARY |
|--------|----------|--------|
| 123    | Jack     | 30000  |
| 142    | Harry    | 60000  |
| 164    | John     | 20000  |
|        | Jackson  | 27000  |

Not allowed as primary key can't contain a NULL value

**3) Referential Integrity Constraints**

- ✓ A referential integrity constraint is specified between two tables.
- ✓ In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

**Example:**

(Table 1)

| EMP_NAME | NAME  | AGE | D_No |
|----------|-------|-----|------|
| 1        | Jack  | 20  | 11   |
| 2        | Harry | 40  | 24   |
| 3        | John  | 27  | 18   |
| 4        | Devil | 38  | 13   |

Foreign key

Not allowed as D\_No 18 is not defined as a Primary key of table 2 and in table 1, D\_No is a foreign key defined

Relationships

(Table 2)

| <u>D_No</u> | D_Location |
|-------------|------------|
| 11          | Mumbai     |
| 24          | Delhi      |
| 13          | Noida      |

Primary Key

**Referential integrity constraint**

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

**Example:** COMPANY database, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a foreign key. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas  $R_1$  and  $R_2$ .

A set of attributes FK in relation schema  $R_1$  is a foreign key of  $R_1$  that references relation  $R_2$  if it satisfies the following rules:

1. Attributes in FK have the same domain(s) as the primary key attributes PK of  $R_2$ ; the attributes FK are said to reference or refer to the relation  $R_2$ .
2. A value of FK in a tuple  $t_1$  of the current state  $r_1(R_1)$  either occurs as a value of PK for some tuple  $t_2$  in the current state  $r_2(R_2)$  or is NULL.

In the former case, we have  $t_1[FK] = t_2[PK]$ , and we say that the tuple  $t_1$  references or refers to the tuple  $t_2$ .

In this definition,  $R_1$  is called the referencing relation and  $R_2$  is the referenced relation. If these two conditions hold, a referential integrity constraint from  $R_1$  to  $R_2$  is said to hold.

**4) Key constraints**

- ✓ Keys are the entity set that is used to identify an entity within its entity set uniquely.
- ✓ An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

**Example:**

| ID   | NAME     | SEMENSTER       | AGE |
|------|----------|-----------------|-----|
| 1000 | Tom      | 1 <sup>st</sup> | 17  |
| 1001 | Johnson  | 2 <sup>nd</sup> | 24  |
| 1002 | Leonardo | 5 <sup>th</sup> | 21  |
| 1003 | Kate     | 3 <sup>rd</sup> | 19  |
| 1002 | Morgan   | 8 <sup>th</sup> | 22  |

Not allowed. Because all row must be unique

**1.2.5 Other Types of Constraints**

**1) Semantic Integrity Constraints:**

Semantic integrity constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose constraint specification language. Examples of such constraints are the salary of an employee should not exceed the salary of the employee’s supervisor and the maximum number of hours an employee can work on all projects per week is 56. Mechanisms called triggers and assertions can be used. In SQL, CREATE ASSERTION and CREATE TRIGGER statements can be used for this purpose.

## 2) Functional Dependency Constraints:

Functional dependency constraint establishes a functional relationship among two sets of attributes X and Y. This constraint specifies that the value of X determines a unique value of Y in all states of a relation; it is denoted as a functional dependency  $X \rightarrow Y$ . We use functional dependencies and other types of dependencies as tools to analyze the quality of relational design and to normalize relations to improve their quality.

### **State constraints (static constraints)**

Define the constraints that a valid state of the database must satisfy

### **Transition constraints (dynamic constraints)**

Define to deal with state changes in the database

## 2. SQL

### Introduction:

SQL was called SEQUEL (Structured English Query Language) and was designed and implemented at IBM Research. The SQL language may be considered one of the major reasons for the commercial success of relational databases. SQL is a comprehensive database language. It has statements for data definitions, queries, and updates. Hence, it is both a DDL and a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++.

### 2.1 Data Definition

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

#### 2.1.1 Schema and Catalog Concepts in SQL

An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement.

**Example:** The following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'.

**CREATE SCHEMA COMPANY AUTHORIZATION Jsmith;**

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.

SQL 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. Schemas within the same catalog can also share certain elements, such as domain definitions.

**2.1.2 The CREATE TABLE Command in SQL**

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and 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.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

```
CREATE TABLE COMPANY.EMPLOYEE ...
```

rather than

```
CREATE TABLE EMPLOYEE ...
```

The relations declared through CREATE TABLE statements are called base tables.

**Examples:**

```
CREATE TABLE EMPLOYEE
( Fname          VARCHAR(15)      NOT NULL,
  Minit          CHAR,
  Lname          VARCHAR(15)   NOT NULL,
  Ssn            CHAR(9)       NOT NULL,
  Bdate         DATE,
  Address        VARCHAR(30),
  Sex            CHAR,
  Salary         DECIMAL(10,2),
  Super_ssn     CHAR(9),
  Dno            INT           NOT NULL,
  PRIMARY KEY (Ssn),
  FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
  FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
```

```

CREATE TABLE DEPARTMENT
  ( Dname          VARCHAR(15)          NOT NULL,
    Dnumber        INT                NOT NULL,
    Mgr_ssn        CHAR(9)           NOT NULL,
    Mgr_start_date DATE,
    PRIMARY KEY (Dnumber),
    UNIQUE (Dname),
    FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );

CREATE TABLE DEPT_LOCATIONS
  ( Dnumber        INT                NOT NULL,
    Dlocation       VARCHAR(15)       NOT NULL,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE PROJECT
  ( Pname          VARCHAR(15)          NOT NULL,
    Pnumber        INT                NOT NULL,
    Plocation       VARCHAR(15),
    Dnum           INT                NOT NULL,
    PRIMARY KEY (Pnumber),
    UNIQUE (Pname),
    FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE WORKS_ON
  ( Essn          CHAR(9)             NOT NULL,
    Pno           INT                NOT NULL,
    Hours         DECIMAL(3,1)       NOT NULL,
    PRIMARY KEY (Essn, Pno),
    FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
    FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );

CREATE TABLE DEPENDENT
  ( Essn          CHAR(9)             NOT NULL,
    Dependent_name VARCHAR(15)       NOT NULL,
    Sex           CHAR,
    Bdate        DATE,
    Relationship   VARCHAR(8),
    PRIMARY KEY (Essn, Dependent_name),
    FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
    
```

### 2.1.3 Attribute Data Types and Domains in SQL

#### Basic Data Types

##### 1) **Numeric Data Types includes**

- integer numbers of various sizes (INTEGER or INT, and SMALLINT
- floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION).
- Formatted numbers can be declared by using

DECIMAL(i,j) or DEC(i,j) or NUMERIC(i,j)

Where i - precision, total number of decimal digits

j - scale, number of digits after the decimal point

## 2) Character String Data Types

- 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

For fixed length strings, a shorter string is padded with blank characters to the right

- **Example:** If the value 'Smith' is for the 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.
- Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents.
- The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G).
- **Example:** CLOB(20M) specifies a maximum length of 20 megabytes.

## 3) Bit-string data types are either of

- Fixed length n BIT(n) or varying length BIT VARYING(n), where n is the maximum number of bits.
- The default for n, the length of a character string or bit string, is 1.
- Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; **Example:** B'10101'
- Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images.
- The maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G)
- **Example:** BLOB(30G) specifies a maximum length of 30 gigabits.

4) A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.

5) The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD.

6) The **TIME** data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.  
Only valid dates and times should be allowed by the SQL implementation.

7) **TIME WITH TIME ZONE** data type includes an additional six positions for specifying the displacement from the standard universal time zone, which is in the range +13:00 to -12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

### Additional Data Types:

- 1) **Timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier.
- 2) **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.

It is possible to specify the data type of each attribute directly or a domain can be declared, and the domain name used with the attribute Specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. 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) for the attributes Ssn and Super\_ssn of EMPLOYEE, Mgr\_ssn of DEPARTMENT, Essn of WORKS\_ON, and Essn of DEPENDENT

## 2.2 Constraints

Basic constraints that can be specified in SQL as part of table creation:

- Key and referential integrity constraints
- Restrictions on attribute domains and NULLs
- Constraints on individual tuples within a relation

### 2.2.1 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. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

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.

```
CREATE TABLE DEPARTMENT
(
  ...,
  Mgr_ssn CHAR(9) NOT NULL DEFAULT '88866555',
  -----
  -----
)
```

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 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 number such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

### 2.2.2 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:

```
Dnumber INT PRIMARY KEY;
```

The **UNIQUE** clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:

```
Dname VARCHAR(15) UNIQUE;
```

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

```
FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(ssn),  
FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber)
```

A referential integrity constraint can be violated when tuples 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, which is known as the **RESTRICT** option.

The schema designer can specify an alternative action to be taken 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**.

- **FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber) ON DELETE SET DEFAULT ON UPDATE CASCADE**
- **FOREIGN KEY (Super\_ssn) REFERENCES EMPLOYEE(Ssn) ON DELETE SET NULL ON UPDATE CASCADE**
- **FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) ON DELETE CASCADE ON UPDATE CASCADE**

In general, the action taken by the DBMS for **SET NULL** or **SET DEFAULT** is the same for both **ON DELETE** and **ON UPDATE**: The value of the affected referencing attributes is changed to **NULL** for **SET NULL** and to the specified default value of the referencing attribute for **SET DEFAULT**.

The action for **CASCADE ON DELETE** is to delete all the referencing tuples whereas the action for **CASCADE ON UPDATE** is to change the value of the referencing foreign key

attribute(s) to the updated (new) primary key value for all the referencing tuples. It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema. As a general rule, the CASCADE option is suitable for “relationship” relation such as WORKS ON: for relations that represent multivalued attributes, such as DEPT\_LOCATIONS; and for relations that represent weak entity types, such as DEPENDENT.

### 2.2.3 Giving Names to Constraints

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.

### 2.2.4 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 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 the managers start date is later than the department creation date.

```
CHECK (Dept_create_date <= Mgr_start_date);
```

## 2.3 Basic Queries for Retrieval in SQL

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

### 2.3.1 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 conditionnal (Boolean) expression that identifies the tuples to be retrieved by the query

#### **Examples:**

- 1) Retrieve date of birth and the address of the employee(s) whose name is ‘John B.Smith’. **SELECT** Bdate, Address

```

FROM EMPLOYEE
WHERE Fname='John' AND Minit='B' AND Lname='Smith';

```

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the projection attributes. The WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition**.

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

```

SELECT Fname, Lname, Address
FROM EMPLOYEE, DEPARTMENT
WHERE Dname='Reaearch' AND Dnumber=Dno;

```

In the WHERE clause, the condition Dname='Reaearch' is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a join condition, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to thevalue of Dno in EMPLOYEE.A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join** 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.

```

SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dname=Dnumber AND Mgr_Ssn=Ssn AND Plocation='Stafford';

```

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

### 2.3.2 Ambiguous Attribute Names, Aliasing, Renaming and Tuple Variables

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

**Example:** Retrieve the name and address of all employees who work for the 'Research' department

```

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

```

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice. For example consider the query: For each employee retrieve them employee's first and last name and the first and last name of his or her immediate supervisor.

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

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

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

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

### 2.3.3 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.

**Example:** Select all EMPLOYEE Ssns and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname in the database.

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

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (\*), which stands for all the attributes. For example, the following query retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5

```
SELECT * FROM EMPLOYEE WHERE Dno=5;
SELECT * FROM EMPLOYEE, DEPARTMENT WHERE Dname='Research'
AND Dno=Dnumber;
SELECT * FROM EMPLOYEE, DEPARTMENT;
```

### 2.3.4 Tables as Sets in SQL

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

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

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.

**Example :** Retrieve the salary of every employee and all distinct salary values

- (a) **SELECT ALL** Salary **FROM** EMPLOYEE;
- (b) **SELECT DISTINCT** Salary **FROM** EMPLOYEE;

| (a)    | (b)    |
|--------|--------|
| Salary | Salary |
| 30000  | 30000  |
| 40000  | 40000  |
| 25000  | 25000  |
| 43000  | 43000  |
| 38000  | 38000  |
| 25000  | 55000  |
| 25000  |        |
| 55000  |        |

SQL has directly incorporated some of the set operations from mathematical set theory, which are also part of relational algebra. There are

- ✓ set union (UNION)
- ✓ set difference (EXCEPT) and
- ✓ set intersection (INTERSECT)

The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result. These set operations apply only to union-compatible relations, so 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.

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

**(SELECT DISTINCT** Pnumber **FROM** PROJECT, DEPARTMENT,

EMPLOYEE **WHERE** Dnum=Dnumber **AND** Mgr\_ssn=Ssn **AND**

**UNION**

**(SELECT DISTINCT** Pnumber **FROM** PROJECT, WORKS\_ON, EMPLOYEE **WHERE** Pnumber=Pno **AND** Essn=Ssn **AND** Lname='Smith');

### 2.3.5 Substring Pattern Matching and Arithmetic Operators several more features of SQL

The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters:

- % replaces an arbitrary number of zero or more
- characters \_ (underscore) replaces a single character

For example, consider the following query: Retrieve all employees whose address is in Houston, Texas

```
SELECT Fname, Lname FROM EMPLOYEE WHERE Address
LIKE '%HoustonTX%';
```

To retrieve all employees who were born during the 1950s, we can use Query

```
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.

#### **Example:**

'AB\_CD\%EF' ESCAPE '\' represents the lateral string, AB\_CD%EF because \ is specified as the escape character. Also, we need a rule to specify apostrophes (') so that it will not be interpreted as ending string.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction ( - ), multiplication (\*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue the following query:

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

**Example:** Retrieve all employees in department 5 whose salary is between \$30,000 and \$40,000.

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

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

### 2.3.6 Ordering of Query Results

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

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

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

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

```
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

## 2.4 INSERT, DELETE, and UPDATE Statements in SQL

### 2.4.1 The INSERT Command

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.

**Example:**

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

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

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. 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.

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

```
U3A: CREATE TABLE WORKS_ON_INFO (  
    Emp_name VARCHAR(15),  
    Proj_name VARCHAR(15),
```

```
Hours_per_week DECIMAL(3,1) );
```

```
U3B: INSERT INTO WORKS_ON_INFO
```

```
(Emp_name, Proj_name, Hours_per_week )
```

```
SELECT E.Lname, P.Pname, W.Hours
```

```
FROM PROJECT P, WORKS_ON W, EMPLOYEE E
```

```
WHERE P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

A table WORKS\_ON\_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B. We can now query WORKS\_ON\_INFO as we would any other relation;

#### **2.4.2 The DELETE Command**

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. The deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL.

**Example:**

```
DELETE FROM EMPLOYEE WHERE Lname='Brown';
```

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.

#### **2.4.3 The UPDATE Command**

The UPDATE command is used to modify attribute values of one or more selected Tuples. 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

```
UPDATE PROJECT SET Plocation='Bellaire', Dnum=5 WHERE Pnumber=10;
```

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.

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, as shown by the following query

```
UPDATE EMPLOYEE
```

```
SET Salary = Salary * 1.1
```

```
WHERE Dno = 5;
```

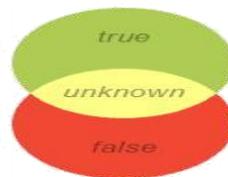
Each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

## 2.5 SQL Advanced Queries: Assertions, Triggers, and Views Three-Valued Logic of SQL:

- SQL uses a three-valued logic: besides true and false, the result of logical expressions can also be unknown.
- SQL's three valued logic is a consequence of supporting null to mark absent data. If a null value affects the result of a logical expression, the result is neither true nor false but unknown.
- The three-valued logic is an integral part of Core SQL and it is followed by every SQL database. Following are the categories:
  - I. Comparisons to NULL
  - II. Logical Operations Involving Unknown
  - III. General Rule: where, having, when, etc.

### I. Comparisons to null:

- The SQL null value basically means “could be anything”. It is therefore impossible to tell whether a comparison to null is true or false.
- That's where the third logical value, unknown, comes in. Unknown means “true or false, depending on the null values”.
- The result of each of the following comparisons is therefore unknown:0  
 $NULL = 1$   
 $NULL <> 1$   
 $NULL > 1$   
 $NULL = NULL$
- Nothing equals null.  
 Not even null equals null because each null could be different.



### II. Logical Operations Involving Unknown:

- In logical connections (and, or), unknown behaves like the null value in comparisons: The result is unknown if it depends on an operand that is unknown.
- The reason is that the result of a logical connection is only unknown if it actually depends on an operand that is unknown.
- **Example:**  
 $(NULL = 1) \text{ OR } (1 = 1)$
- Although the comparison to null makes the first operand of the operation unknown, the total result is still true because or operations are true as soon as any operand is true.
- In the example above you can assume the values 0 and 1 instead of null to make the result of the first operand false and true respectively.

- But the result of the complete expression is true in both cases — it does not depend on the value you assume for null.
- A similar case applies to the and operator: and connections are false as soon as any operand is false.
- The result of the following expression is therefore false: (NULL = 1) AND (0 = 1)
- In all other cases, any unknown operand for not, and, and or causes the logical operation to return unknown.

**III. General Rule: where, having, when, etc.**

- It is not enough that a condition is not false.
- The result of the following query is therefore always the empty set:

**SELECT col FROM t  
WHERE col = NULL**

- The result of the equals comparison to null is always unknown. The where clause thus rejects all rows.
- Use the null predicate to search for null values:

**WHERE col IS NULL**

- Odd Consequence: not in (null, ...) is never true
- Consider this example:

**WHERE 1 NOT IN (NULL)**

- ✓ True
- ✓ Unknown
- ✓ False

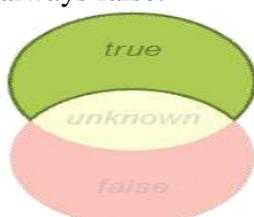
- Two values for null that make the expression true and false respectively. Let's take 0 and 1.
- For 0, the expressions becomes 1 NOT IN (0), which is true.
- For 1, the expression becomes 1 NOT IN (1), which is clearly false.
- The result of the original expression is therefore unknown, because it changes if null is replaced by different values.
- Result of not in predicates that contain a null value is never true:

**WHERE 1 NOT IN (NULL, 2)**

- This expression is again unknown because substituting different values for null (e.g. 0 and 1) still influences the result.
- Not in predicates that contain a null value can be false:

**WHERE 1 NOT IN (NULL, 1)**

- No matter which value you substitute for the null (0, 1 or any other value) the result is always false.



**Nested Queries:**

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. We will use STUDENT, COURSE, STUDENT\_COURSE tables for understanding nested queries.

STUDENT Table data:

| S_ID | S_NAME | S_ADDRESS | S_PHONE    | S_AGE |
|------|--------|-----------|------------|-------|
| S1   | RAM    | DELHI     | 9455123451 | 18    |
| S2   | RAMESH | GURGAON   | 9652431543 | 18    |
| S3   | SUJIT  | ROHTAK    | 9156253131 | 20    |
| S4   | SURESH | DELHI     | 9156768971 | 18    |

COURSE table data:

| C_ID | C_NAME      |
|------|-------------|
| C1   | DSA         |
| C2   | Programming |
| C3   | DBMS        |

STUDENT\_COURSE table data:

| S_ID | C_ID |
|------|------|
| S1   | C1   |
| S1   | C3   |
| S2   | C1   |
| S3   | C2   |
| S4   | C2   |
| S4   | C3   |

There are mainly two types of nested queries:

1. Independent Nested Queries
2. Co-related Nested Queries

**I. Independent Nested Queries:**

- In independent nested queries, query execution starts from innermost query to outermost queries.
- The execution of inner query is independent of outer query, but the result of inner query is used in execution of outer query.
- Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.
- **IN:** If we want to find out S\_ID who are enrolled in C\_NAME ‘DSA’ or ‘DBMS’, we can write it with the help of independent nested query and IN operator.  
From COURSE table, we can find out C\_ID for C\_NAME ‘DSA’ or DBMS’ and we can use these C\_IDs for finding S\_IDs from STUDENT\_COURSE TABLE.

- **STEP 1:** Finding C\_ID for C\_NAME = 'DSA' or 'DBMS'  
**Select C\_ID from COURSE where C\_NAME = 'DSA' or C\_NAME = 'DBMS'**
- **STEP 2:** Using C\_ID of step 1 for finding S\_ID  
**Select S\_ID from STUDENT\_COURSE where C\_ID IN  
(SELECT C\_ID from COURSE where C\_NAME = 'DSA' or  
C\_NAME='DBMS');**
- The inner query will return a set with members C1 and C3 and outer query will return those S\_IDs for which C\_ID is equal to any member of set (C1 and C3 in this case). So, it will return S1, S2 and S4.
- **Note:** If we want to find out names of STUDENTs who have either enrolled in 'DSA' or 'DBMS', it can be done as:  
**Query:**  
**Select S\_NAME from STUDENT where S\_ID IN (Select  
S\_ID from STUDENT\_COURSE where C\_ID IN  
(SELECT C\_ID from COURSE where  
C\_NAME='DSA' or C\_NAME='DBMS'));**

### **I. Independent Nested Queries:**

- **NOT IN:** If we want to find out S\_IDs of STUDENTs who have neither enrolled in 'DSA' nor in 'DBMS', it can be done as:  
**Select S\_ID from STUDENT where S\_ID NOT IN  
(Select S\_ID from STUDENT\_COURSE where C\_ID IN  
(SELECT C\_ID from COURSE where  
C\_NAME='DSA' or C\_NAME='DBMS'));**
- The innermost query will return a set with members C1 and C3. Second inner query will return those S\_IDs for which C\_ID is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those S\_IDs where S\_ID is not a member of set (S1, S2 and S4). So it will return S3.

### **II. Co-related Nested Queries:**

- The output of inner query depends on the row which is being currently executed in outer query.  
**Example:** If we want to find out S\_NAME of STUDENTs who are enrolled in C\_ID 'C1', it can be done with the help of co-related nested query as:  
**Select S\_NAME from STUDENT S where EXISTS  
( select \* from STUDENT\_COURSE SC where S.S\_ID=SC.S\_ID and  
SC.C\_ID='C1');**
- For each row of STUDENT S, it will find the rows from STUDENT\_COURSE where S.S\_ID = SC.S\_ID and SC.C\_ID='C1'.
- If for a S\_ID from STUDENT S, at least a row exists in STUDENT\_COURSE SC with C\_ID='C1', then inner query will return true and corresponding S\_ID will be returned as output.

**EXISTS:**

- The EXISTS condition in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not.
- The result of EXISTS is a boolean value True or False.
- It can be used in a SELECT, UPDATE, INSERT or DELETE statement.

- **Syntax:**

```

SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name(s)
FROM table_name
WHERE condition);
    
```

- **Examples:**

Consider the following two relation “Customers” and “Orders”.

**Customers**

| customer_id | lname   | fname   | website |
|-------------|---------|---------|---------|
| 401         | Singh   | Dolly   | abc.com |
| 402         | Chauhan | Anuj    | def.com |
| 403         | Kumar   | Niteesh | ghi.com |
| 404         | Gupta   | Shubham | jkl.com |
| 405         | Walecha | Divya   | abc.com |
| 406         | Jain    | Sandeep | jkl.com |
| 407         | Mehta   | Rajiv   | abc.com |
| 408         | Mehra   | Anand   | abc.com |

**Orders**

| order_id | c_id | order_date |
|----------|------|------------|
| 1        | 407  | 2017-03-03 |
| 2        | 405  | 2017-03-05 |
| 3        | 408  | 2017-01-18 |
| 4        | 404  | 2017-02-05 |

**Queries:**

**I. Using EXISTS condition with SELECT statement**

To fetch the first and last name of the customers who placed at least one order.

```

SELECT fname, lname
FROM Customers
WHERE EXISTS (SELECT *
FROM Orders
WHERE Customers.customer_id = Orders.c_id);
    
```

**Output:**

| fname   | lname   |
|---------|---------|
| Shubham | Gupta   |
| Divya   | Walecha |
| Rajiv   | Mehta   |
| Anand   | Mehra   |

**II. Using NOT with EXISTS**

Fetch last and first name of the customers who has not placed any order.

```

SELECT lname, fname
FROM Customer
WHERE NOT EXISTS (SELECT *
                  FROM Orders
                  WHERE Customers.customer_id = Orders.c_id);
    
```

**Output:**

| lname   | fname   |
|---------|---------|
| Singh   | Dolly   |
| Chauhan | Anuj    |
| Kumar   | Niteesh |
| Jain    | Sandeep |

**III. Using EXISTS condition with DELETE statement**

Delete the record of all the customer from Order Table whose last name is ‘Mehra’.

```

DELETE
FROM Orders
WHERE EXISTS (SELECT *
             FROM customers
             WHERE Customers.customer_id = Orders.cid
             AND Customers.lname = 'Mehra');
SELECT * FROM Orders;
    
```

**Output:**

| order_id | c_id | order_date |
|----------|------|------------|
| 1        | 407  | 2017-03-03 |
| 2        | 405  | 2017-03-05 |
| 4        | 404  | 2017-02-05 |

**IV. Using EXISTS condition with UPDATE statement**

Update the lname as ‘Kumari’ of customer in Customer Table whose customer\_id is 401.

```

UPDATE Customers
SET lname = 'Kumari'
WHERE EXISTS (SELECT *
             FROM Customers
             WHERE customer_id = 401);
    
```

WHERE customer\_id = 401);  
 SELECT \* FROM Customers;

**Output:**

| customer_id | lname   | fname   | website |
|-------------|---------|---------|---------|
| 401         | Kumari  | Dolly   | abc.com |
| 402         | Chauhan | Anuj    | def.com |
| 403         | Kumar   | Niteesh | ghi.com |
| 404         | Gupta   | Shubham | jkl.com |
| 405         | Walecha | Divya   | abc.com |
| 406         | Jain    | Sandeep | jkl.com |
| 407         | Mehta   | Rajiv   | abc.com |
| 408         | Mehra   | Anand   | abc.com |

**Aggregate functions:**

Aggregate functions return single values by performing action of a group of values.

**Example:** Emp

| Emp_id | Name   | Salary | Age |
|--------|--------|--------|-----|
| 350    | Mark   | 55000  | 28  |
| 351    | Steve  | 60000  | 30  |
| 352    | Roser  | 60000  | 31  |
| 353    | Joseph | 75000  | 34  |
| 354    | Avlin  | 90000  | 32  |

- **Sum ()**: this function calculates a sum of a group of values.

Syntax: sum(col \_name)

Example:

Select sum(salary) from Emp;

Output:

340000

- **Average ()**: this function returns the average of group of values.

Syntax: average(col \_name)

Example:

Select avg(salary) from Emp;

Output:

68000

- **Max ()**: this function returns the highest value from group of values. Syntax: max(col \_name)

Example:

Select max(salary) from Emp;

Output:

90000

- **Min ()**: this function returns the least value from group of values.

Syntax: min(col \_name)

Example:

Select min(salary) from Emp;

Output:

55000

- **Count ():** this function counts the number of values in a group. Syntax: count(col \_name)

Example:

Select count(name) from Emp where salary =

60000; Output:

2

**Group By:**

- The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.
- Important Points:
  - ✓ GROUP BY clause is used with the SELECT statement.
  - ✓ In the query, GROUP BY clause is placed after the WHERE clause.
  - ✓ In the query, GROUP BY clause is placed before ORDER BY clause if used any.

- **Syntax:**

**SELECT column1, function\_name(column2)**

**FROM table\_name WHERE condition GROUP BY column1, column2 ORDER BY column1, column2;**

function\_name: Name of the function used for example, SUM() , AVG().

table\_name: Name of the table.

condition: Condition used.

**Sample data tables to use in Query:**

| Employee |         |        |     |
|----------|---------|--------|-----|
| SI NO    | NAME    | SALARY | AGE |
| 1        | Harsh   | 2000   | 19  |
| 2        | Dhanraj | 3000   | 20  |
| 3        | Ashish  | 1500   | 19  |
| 4        | Harsh   | 3500   | 19  |
| 5        | Ashish  | 1500   | 19  |

**Example:**

- Group By single column: Group By single column means, to place all the rows with same value of only that particular column in one group.
- Consider the query as shown below:

**SELECT NAME, SUM(SALARY) FROM  
Employee GROUP BY NAME;**

**OUTPUT:**

| NAME    | SALARY |
|---------|--------|
| Ashish  | 3000   |
| Dhanraj | 3000   |
| Harsh   | 5500   |

As you can see in the above output, the rows with duplicate NAMES are grouped under same NAME and their corresponding SALARY is the sum of the SALARY of duplicate rows. The SUM() function of SQL is used here to calculate the sum.

**Group By multiple columns:** Group by multiple column is say for example, GROUP BY column1, column2. This means to place all the rows with same values of both the columns column1 and column2 in one group.

Query:

**SELECT SUBJECT, YEAR, Count(\*) FROM  
Student GROUP BY SUBJECT, YEAR;**

**Output:**

| SUBJECT     | YEAR | Count |
|-------------|------|-------|
| English     | 1    | 3     |
| English     | 2    | 2     |
| Mathematics | 1    | 2     |

As you can see in the above output the students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

**HAVING Clause:**

- HAVING clause can be used to place conditions to decide which group will be the part of final result-set.
- We can not use the aggregate functions like SUM(), COUNT() etc. with WHERE clause.
- So we have to use HAVING clause if we want to use any of these functions in the conditions.
- As you can see in the output only one group out of the three groups appears in the result-set as it is the only group where sum of SALARY is greater than 3000.
- So we have used HAVING clause here to place this condition as the condition is required to be placed on groups not columns.

**Syntax:**

**SELECT column1, function\_name(column2)  
FROM table\_name**

**WHERE condition**  
**GROUP BY column1, column2**  
**HAVING condition**  
**ORDER BY column1, column2;**

function\_name: Name of the function used for example, SUM() , AVG().

table\_name: Name of the table.

condition: Condition used.

**SELECT NAME, SUM(SALARY) FROM Employee**  
**GROUP BY NAME**  
**HAVING SUM(SALARY)>3000;**

**Output:**

| NAME  | SUM(SALARY) |
|-------|-------------|
| HARSH | 5500        |

**ORDER BY:**

- The ORDER BY statement in SQL is used to sort the fetched data in either ascending or descending according to one or more columns.
- By default ORDER BY sorts the data in ascending order.
- We can use the keyword DESC to sort the data in descending order and the keyword ASC to sort in ascending order.
- Sort according to one column
- To sort in ascending or descending order we can use the keywords ASC or DESC respectively.

**Syntax:**

**SELECT \* FROM table\_name ORDER BY column\_name ASC|DESC;**

Where

table\_name: name of the table.

column\_name: name of the column according to which the data is needed to be arranged.

ASC: to sort the data in ascending order.

DESC: to sort the data in descending order.

|: use either ASC or DESC to sort in ascending or descending order

**Sort according to multiple columns:**

To sort in ascending or descending order we can use the keywords ASC or DESC respectively.

To sort according to multiple columns, separate the names of columns by the (,) operator.

**Syntax:**

**SELECT \* FROM table\_name ORDER BY column1 ASC|DESC, column2 ASC|DESC;**

| ROLL_NO | NAME     | ADDRESS   | PHONE      | Age |
|---------|----------|-----------|------------|-----|
| 1       | HARSH    | DELHI     | XXXXXXXXXX | 18  |
| 2       | PRATIK   | BIHAR     | XXXXXXXXXX | 19  |
| 3       | RIYANKA  | SILIGURI  | XXXXXXXXXX | 20  |
| 4       | DEEP     | RAMNAGAR  | XXXXXXXXXX | 18  |
| 5       | SAPTARHI | KOLKATA   | XXXXXXXXXX | 19  |
| 6       | DHANRAJ  | BARABAJAR | XXXXXXXXXX | 20  |
| 7       | ROHIT    | BALURGHAT | XXXXXXXXXX | 18  |
| 8       | NIRAJ    | ALIPUR    | XXXXXXXXXX | 19  |

Now consider the above database table and find the results of different queries.

Sort according to a single column:

In this example, we will fetch all data from the table Student and sort the result in descending order according to the column ROLL\_NO.

**Query:**

```
SELECT * FROM Student ORDER BY ROLL_NO DESC;
```

**Output:**

| ROLL_NO | NAME     | ADDRESS   | PHONE      | Age |
|---------|----------|-----------|------------|-----|
| 8       | NIRAJ    | ALIPUR    | XXXXXXXXXX | 19  |
| 7       | ROHIT    | BALURGHAT | XXXXXXXXXX | 18  |
| 6       | DHANRAJ  | BARABAJAR | XXXXXXXXXX | 20  |
| 5       | SAPTARHI | KOLKATA   | XXXXXXXXXX | 19  |
| 4       | DEEP     | RAMNAGAR  | XXXXXXXXXX | 18  |
| 3       | RIYANKA  | SILIGURI  | XXXXXXXXXX | 20  |
| 2       | PRATIK   | BIHAR     | XXXXXXXXXX | 19  |
| 1       | HARSH    | DELHI     | XXXXXXXXXX | 18  |

In the above example, if we want to sort in ascending order we have to use ASC in place of DESC.

**Sort according to multiple columns:**

- In this example we will fetch all data from the table Student and then sort the result in ascending order first according to the column Age. And then in descending order according to the column ROLL\_NO.
- Note:  
ASC is the default value for the ORDER BY clause. So, if we don't specify anything after the column name in the ORDER BY clause, the output will be sorted in ascending order by default.

**Query:**

```
SELECT * FROM Student ORDER BY Age ASC, ROLL_NO DESC;
```

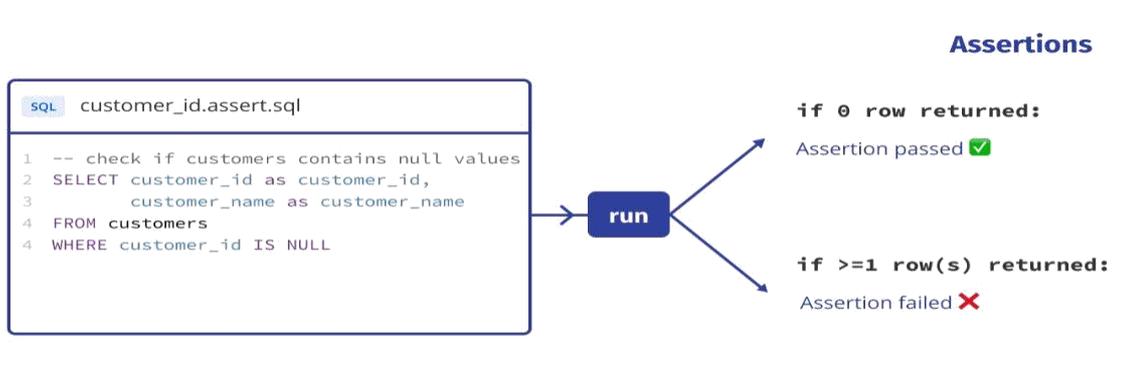
**Output:**

| ROLL_NO | NAME     | ADDRESS   | PHONE        | Age |
|---------|----------|-----------|--------------|-----|
| 7       | ROHIT    | BALURGHAT | XXXXXXXXXXXX | 18  |
| 4       | DEEP     | RAMNAGAR  | XXXXXXXXXXXX | 18  |
| 1       | HARSH    | DELHI     | XXXXXXXXXXXX | 18  |
| 8       | NIRAJ    | ALIPUR    | XXXXXXXXXXXX | 19  |
| 5       | SAPTARHI | KOLKATA   | XXXXXXXXXXXX | 19  |
| 2       | PRATIK   | BIHAR     | XXXXXXXXXXXX | 19  |
| 6       | DHANRAJ  | BARABAJAR | XXXXXXXXXXXX | 20  |
| 3       | RIYANKA  | SILIGURI  | XXXXXXXXXXXX | 20  |

In the above output, we can see that first the result is sorted in ascending order according to Age. There are multiple rows of having the same Age. Now, sorting further this result-set according to ROLL\_NO will sort the rows with the same Age according to ROLL\_NO in descending order.

**Assertions:**

- When a constraint involves 2 (or) more tables, the table constraint mechanism is sometimes hard and results may not come as expected.
- To cover such situation SQL supports the creation of assertions that are constraints not associated with only one table.
- An assertion statement should ensure a certain condition will always exist in the database. DBMS always checks the assertion whenever modifications are done in the corresponding table.
- A data assertion is a query that looks for problems in a dataset. If the query returns any rows then the assertion fails.
- Data assertions are defined this way because it's much easier to look for problems rather than the absence of them.
- It also means that assertion queries can themselves be used to quickly inspect the data causing the assertion to fail - making it easy to diagnose and fix the problem.



**Checking field values:****Example:**

- Assume that there is a database.customers table containing information about customers in the database.
- Some checks that we might want to verify on the table's contents include:
  - ✓ The field email\_address is always set
  - ✓ The field customer\_type is one of "business" or "individual"
- The following simple query will return any rows violating these rules:

```
SELECT customer_id  
FROM database.customers  
WHERE email_address IS NULL  
OR NOT customer_type IN ("business", "individual")
```

**Checking for unique fields:**

We may also want to run checks across more than one row.

**Example:** Verify that the customer\_id field is unique.

A query like the following will return any duplicate customer\_id values:

```
SELECT  
customer_id,  
SUM(1) AS count  
FROM database.customers  
GROUP BY 1  
HAVING count > 1
```

**Combining multiple assertions into a single query:**

We can combine all of the above into a single query to quickly find any customer\_id value violating one of our rules using UNION ALL:

```
SELECT customer_id, "missing_email" AS reason  
FROM database.customers WHERE  
email_address IS NULL  
UNION ALL  
SELECT customer_id, "invalid_customer_type" AS reason WHERE not customer_type  
in ("business", "individual")  
FROM database.customers  
UNION ALL  
SELECT customer_id, "duplicate_id" AS reason  
FROM (SELECT customer_id, SUM(1) AS count FROM database.customers GROUP  
BY 1)  
WHERE count > 1
```

**Triggers:**

- A trigger is a database object that is associated with the table, it will be activated when a defined action is executed for the table. In another way;
- A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs.

- For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.
- The trigger can be executed when we run the following statements:
  1. INSERT
  2. UPDATE
  3. DELETE
- And it can be invoked before or after the event.

**Syntax –**

```
create trigger [trigger_name]
[before | after]
{insert | update |
delete} on [table_name]
[for each row]
[trigger_body]
```

**Explanation of syntax for Trigger:**

**create trigger [trigger\_name]:**

Creates or replaces an existing trigger with the trigger\_name.

**[before | after]:** This specifies when the trigger will be executed.

{insert | update | delete}: This specifies the DML operation.

on **[table\_name]:** This specifies the name of the table associated with the trigger.

**[for each row]:** This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.

**[trigger\_body]:** This provides the operation to be performed as trigger is fired

**BEFORE and AFTER of Trigger:**

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

**Example:**

- Given Student Report Database, in which student marks assessment is recorded.
- In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.
- Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used. Suppose the database Schema shown below is considered.

```
mysql> desc Student;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| tid   | int(4)        | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30)   | YES  |     | NULL    |                |
| subj1 | int(2)        | YES  |     | NULL    |                |
| subj2 | int(2)        | YES  |     | NULL    |                |
| subj3 | int(2)        | YES  |     | NULL    |                |
| total | int(3)        | YES  |     | NULL    |                |
| per   | int(3)        | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

**Example:**

SQL Trigger to problem statement.

```
create trigger stud_marks
before INSERT on Student for each row set
```

```
Student.total = Student.subj1 + Student.subj2 + Student.subj3,
```

```
Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)
```

```
mysql> select * from Student;
```

```
+-----+-----+-----+-----+-----+-----+
| tid | name | subj1 | subj2 | subj3 | total | per |
+-----+-----+-----+-----+-----+-----+-----+
| 100 | ABCDE | 20 | 20 | 20 | 60 | 36 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

In this way trigger can be creates and executed in the databases.

**Views:**

**Creating Views:**

We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables.

**Syntax:**

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

**Syntax explanation:**

view\_name: Name for the View

table\_name: Name of the table

condition: Condition to select rows

Let us see the data from the Sample Tables:

StudentDetails:

| S_ID | NAME    | ADDRESS   |
|------|---------|-----------|
| 1    | Harsh   | Kolkata   |
| 2    | Ashish  | Durgapur  |
| 3    | Pratik  | Delhi     |
| 4    | Dhanraj | Bihar     |
| 5    | Ram     | Rajasthan |

StudentMarks:

| ID | NAME    | MARKS | AGE |
|----|---------|-------|-----|
| 1  | Harsh   | 90    | 19  |
| 2  | Suresh  | 50    | 20  |
| 3  | Pratik  | 80    | 19  |
| 4  | Dhanraj | 95    | 21  |
| 5  | Ram     | 85    | 18  |

**Examples for Creating Views:**

**I. Creating View from a single table:**

In this example we will create a View named DetailsView from the table StudentDetails.

**Query:**

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

| NAME    | ADDRESS  |
|---------|----------|
| Harsh   | Kolkata  |
| Ashish  | Durgapur |
| Pratik  | Delhi    |
| Dhanraj | Bihar    |

**Examples:**

In this example, we will create a view named StudentNames from the table StudentDetails.

**Query:**

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

If we now query the view as,

```
SELECT * FROM StudentNames;
```

**Output:**

| S_ID | NAMES   |
|------|---------|
| 2    | Ashish  |
| 4    | Dhanraj |
| 1    | Harsh   |
| 3    | Pratik  |
| 5    | Ram     |

**II. Creating View from multiple tables:**

In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks.

To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

**Query:**

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

**Output:**

| NAME    | ADDRESS   | MARKS |
|---------|-----------|-------|
| Harsh   | Kolkata   | 90    |
| Pratik  | Delhi     | 80    |
| Dhanraj | Bihar     | 95    |
| Ram     | Rajasthan | 85    |

**DELETING VIEWS:**

SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

**Syntax:**

```
DROP VIEW view_name;
```

view\_name: Name of the View which we want to delete.

**Example:** if we want to delete the View MarksView, we can do this as:

```
DROP VIEW MarksView;
```

**UPDATING VIEWS:**

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- The view should be created from a single table.
- If the view is created using multiple tables then we will not be allowed to update the view.

We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.

**Syntax:**

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

For example, if we want to update the view MarksView and add the field AGE to this View from StudentMarks Table, we can do this as:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS, StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

### **Output:**

| NAME    | ADDRESS   | MARKS | AGE |
|---------|-----------|-------|-----|
| Harsh   | Kolkata   | 90    | 19  |
| Pratik  | Delhi     | 80    | 19  |
| Dhanraj | Bihar     | 95    | 21  |
| Ram     | Rajasthan | 85    | 18  |

### **Inserting a row in a view:**

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

### **Syntax:**

```
INSERT INTO view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);
```

view\_name: Name of the View

### **Example:**

In the below example we will insert a new row in the View DetailsView which we have created above in the example of “creating views from a single table”.

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

**Output:**

| NAME    | ADDRESS  |
|---------|----------|
| Harsh   | Kolkata  |
| Ashish  | Durgapur |
| Pratik  | Delhi    |
| Dhanraj | Bihar    |
| Suresh  | Gurgaon  |

**Deleting a row from a View:**

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view.

Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.

**Syntax:**

**DELETE FROM view\_name  
WHERE condition;**

view\_name: Name of view from where we want to delete rows

condition: Condition to select rows **Example:**

In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

**DELETE FROM DetailsView  
WHERE NAME="Suresh";**

If we fetch all the data from DetailsView now as,

**SELECT \* FROM DetailsView;**

**Output:**

| NAME    | ADDRESS  |
|---------|----------|
| Harsh   | Kolkata  |
| Ashish  | Durgapur |
| Pratik  | Delhi    |
| Dhanraj | Bihar    |

**WITH CHECK OPTION:**

- ✓ The WITH CHECK OPTION clause in SQL is a very useful clause for views.
- ✓ It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.
- ✓ The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- ✓ If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

**Example:**

In the below example we are creating a View SampleView from StudentDetails Table with

```
WITH CHECK OPTION clause.  
CREATE VIEW SampleView AS  
SELECT S_ID, NAME  
FROM StudentDetails  
WHERE NAME IS NOT NULL  
WITH CHECK OPTION;
```

In this View if we now try to insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL.

For example, though the View is updatable but then also the below query for this View is not valid:

```
INSERT INTO SampleView(S_ID)  
VALUES(6);
```

NOTE: The default value of NAME column is null.

### **3. Formal Relational Languages:**

#### **3.1 Introduction**

Relational algebra is the basic set of operations for the relational model. These operations enable a user to specify basic retrieval requests as relational algebra expressions. The result of an operation is a new relation, which may have been formed from one or more input relations. The relational algebra is very important for several reasons

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

**Relation Algebra** is a procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result. Six basic operators are:

1. select:  $\sigma$
2. project:  $\Pi$
3. union:  $\cup$
4. set difference:  $-$
5. Cartesian product:  $\times$
6. rename:  $\rho$

#### **2.2 Unary Relational Operations: SELECT and PROJECT**

##### **2.2.1 The SELECT Operation**

The SELECT operation denoted by ( $\sigma$ ) is used to select a subset of the tuples from a relation based on a selection condition. The selection condition acts as a filter that keeps only

those tuples that satisfy a qualifying condition. Alternatively, we can consider the SELECT operation to restrict the tuples in a relation to only those tuples that satisfy the condition.

The SELECT operation can also be visualized as a horizontal partition of the relation into two sets of tuples those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded.

In general, the select operation is denoted by

$$\langle \text{select condition} \rangle (R)$$

where,

- the symbol  $\sigma$  is used to denote the select operator
- the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
- tuples that make the condition true are selected
  - appear from the result of the operation
- tuples that make the condition false are filtered out
  - discarded from the result of the operation

The Boolean expression specified in  $\langle \text{selection condition} \rangle$  is made up of a number of clauses of the form:

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{constant value} \rangle$  or

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{attribute name} \rangle$

where

$\langle \text{attribute name} \rangle$  is the name of an attribute of R,

$\langle \text{comparison op} \rangle$  is one of the operators  $\{=, <, >, \leq, \geq, \neq\}$

$\langle \text{constant value} \rangle$  is a constant value from the attribute domain

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

- **The select operation selects tuples that satisfy a given predicate.**
- **Notation:**  $\sigma_p(r)$   
p is called the selection predicate
- **Example:** select those tuples of the instructor relation where the instructor is in the “Physics” department.
- **Query:**  
 $\sigma_{\text{dept\_name}=\text{“Physics”}}(\text{instructor})$
- **Result:**

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 22222     | Einstein    | Physics          | 95000         |
| 33456     | Gold        | Physics          | 87000         |

**Examples:**

- 1) Select the EMPLOYEE tuples whose department number is 4.

$$\sigma_{DNO=4}(\text{EMPLOYEE})$$

- 2) Select the employee tuples whose salary is greater than \$30,000.

$$\sigma_{SALARY > 30,000}(\text{EMPLOYEE})$$

- 3) 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

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

The result of a SELECT operation can be determined as follows:

- The <selection condition> is applied independently to each individual tuple t in R
- If the condition evaluates to TRUE, then tuple t is selected. All the selected tuples appear in the result of the SELECT operation
- The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:
  - (cond1 AND cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
  - (cond1 OR cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
  - (NOT cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is unary; that is, it is applied to a single relation. The degree of the relation resulting from a SELECT operation is the same as the degree of R. The number of tuples in the resulting relation is always less than or equal to the number of tuples in R. That is,

$$|\sigma_C(R)| \leq \text{for any condition } C$$

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

The SELECT operation is commutative; that is,

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(R)) = \sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond1} \rangle}(R))$$

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

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(\dots(\sigma_{\langle \text{condn} \rangle}(R))\dots)) = \sigma_{\langle \text{cond1} \rangle \text{ AND } \langle \text{cond2} \rangle \text{ AND } \dots \text{ AND } \langle \text{condn} \rangle}(R)$$

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

**σ<sub>Dno=4 AND Salary>25000</sub> (EMPLOYEE)**

would do the following SQL query:

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

**2.2.2 The PROJECT Operation**

The PROJECT operation denoted by  $\pi$  selects certain columns from the table and discards the other columns. Used when we are interested in only certain attributes of a relation. 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
- the other contains the discarded columns

The general form of the PROJECT operation is

**<attribute list>(R)**

where

$\pi$  (pi) - symbol used to represent the PROJECT operation,

<attributelist> - 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>

- **Project Operation is a unary operation that returns its argument relation, with certain attributes left out.**
- **Notation:**  $\pi_{A_1, A_2, A_3, \dots, A_k}(r)$   
where  $A_1, A_2, \dots, A_k$  are attribute names and  $r$  is a relation name.
- The result is defined as the relation of  $k$  columns obtained by erasing the columns that are not listed.
- Duplicate rows removed from result, since relations are sets.

**Example:** eliminate the dept\_name attribute of instructor

**Query:** ID, name, salary (instructor)

**Result:**

| <i>ID</i> | <i>name</i> | <i>salary</i> |
|-----------|-------------|---------------|
| 10101     | Srinivasan  | 65000         |
| 12121     | Wu          | 90000         |
| 15151     | Mozart      | 40000         |
| 22222     | Einstein    | 95000         |
| 32343     | El Said     | 60000         |
| 33456     | Gold        | 87000         |
| 45565     | Katz        | 75000         |
| 58583     | Califeri    | 62000         |
| 76543     | Singh       | 80000         |
| 76766     | Crick       | 72000         |
| 83821     | Brandt      | 92000         |
| 98345     | Kim         | 80000         |

**Example:**

- 1) To list each employee’s first and last name and salary we can use the PROJECT operation as follows:

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

If the attribute list includes only nonkey attributes of R, duplicate tuples are likely to occur. The result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as duplicate elimination. For example, consider the following PROJECT operation:

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

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

The tuple <'F', 25000> appears only once in the resulting relation even though this combination of values appears twice in the EMPLOYEE relation.

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R. Commutativity does not hold on PROJECT

$$\pi_{\langle list1 \rangle}(\pi_{\langle list2 \rangle}(R)) \neq \pi_{\langle list1 \rangle}(R)$$

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression.

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

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

would correspond to the following SQL query:

**SELECT DISTINCT gender, Salary FROM  
EMPLOYEE;** Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a relational-algebra expression.
- Consider the query -- Find the names of all instructors in the Physics department.  
name( dept\_name = "Physics" (instructor))
- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

**2.2.3 Sequences of Operations and the RENAME Operation**

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ, is provided for that purpose

- The expression:

$$\rho_x(E)$$

returns the result of expression E under the name x

- Another form of the rename operation:

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

For most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single relational algebra expression by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

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

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

Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

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

$$RESULT \leftarrow \pi_{Fname, Lname, Salary}(DEP5\_EMPS)$$

We can also use this technique to rename the attributes in the intermediate and result relations. To rename the attributes in a relation, we simply list the new attribute names in parentheses.

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

$$R(\text{First\_name, Last\_name, Salary}) \leftarrow \pi_{Fname, Lname, Salary}(TEMP)$$

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  |

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

We can also define a formal RENAME operation which can rename either the relation name or the attribute names, or both as a unary operator.

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

|  |  |
|--|--|
| 1. $\rho_{S(B_1, B_2, \dots, B_n)}(R)$ | $\rho$ (rho) – RENAME operator               |
| 2. $\rho_S(R)$                         | S – new relation name                        |
| 3. $\rho_{(B_1, B_2, \dots, B_n)}(R)$  | $B_1, B_2, \dots, B_n$ - new attribute names |

The first expression renames both the relation and its attributes. Second renames the relation only and the third renames the attributes only. If the attributes of R are (A1, A2, ..., An) in that order, then each Ai is renamed as Bi.

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,
```

**3.1 Relational Algebra Operations from Set Theory:**

**3.1.1 The UNION, INTERSECTION and MINUS Operations**

- **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.
- **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):** The result of this operation, denoted by R S, is a relation that includes all tuples that are in R but not in S.

**Example:** Consider the the following two relations: STUDENT & INSTRUCTOR

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    |

STUDENT  $\cup$  INSTRUCTOR

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

STUDENT  $\cap$  INSTRUCTOR

| Fn     | Ln   |
|--------|------|
| Susan  | Yao  |
| Ramesh | Shah |

STUDENT-INSTRUCTOR

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

INSTRUCTOR-STUDENT

| Fname   | Lname   |
|---------|---------|
| John    | Smith   |
| Ricardo | Browne  |
| Francis | Johnson |

**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.

**DEPT5\_EMP** ←  $\sigma_{Dno=5}(EMPLOYEE)$

**RESULT1** ←  $Ssn(DEPT5\_EMPS)$

**RESULT2(Ssn)** ←  $Super\_Ssn(DEPT5\_EMPS)$   
RESULT ← RESULT1  $\cup$  RESULT2

**EMPLOYEE**

| Fname    | Minit | Lname   | Ssn       | Bdate      | Address                  | gender | 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   |

**RESULT1**

| Ssn       |
|-----------|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |

**RESULT2**

| Ssn       |
|-----------|
| 333445555 |
| 888665555 |

**RESULT**

| Ssn       |
|-----------|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |
| 888665555 |

Single relational algebra expression:

$$\mathbf{Result} \leftarrow \pi_{Ssn}(\sigma_{Dno=5}(\mathbf{EMPLOYEE})) \cup \pi_{Ssn}(\sigma_{Dno=4}(\mathbf{EMPLOYEE}))$$

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

Two relations  $R(A_1, A_2, \dots, A_n)$  and  $S(B_1, B_2, \dots, B_n)$  are said to be union compatible (or type compatible) if they have the same degree  $n$  and if  $dom(A_i) = dom(B_i)$  for  $1 \leq i \leq n$ . This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

Both UNION and INTERSECTION are commutative operations; that is,

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

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

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

The MINUS operation is not commutative; that is,

in general,

**R-S ≠ S-R**

INTERSECTION can be expressed in terms of union and set difference as follows,

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

In SQL, there are three operations UNION, INTERSECT, and EXCEPT that correspond to the set operations

**Union Operation:**

- The union operation allows us to combine two relations
  - Notation: r ∪ s  
For r, s to be valid.
  - 1. r, s must have the same arity (same number of attributes)
  - 2. The attribute domains must be compatible (example: 2<sup>nd</sup> column of r deals with the same type of values as does the 2<sup>nd</sup> column of s)

Result of:

$$\Pi_{course\_id} (\sigma_{semester="Fall" \wedge year=2017} (section)) \cup \Pi_{course\_id} (\sigma_{semester="Spring" \wedge year=2018} (section))$$

| course_id |
|-----------|
| CS-101    |
| CS-315    |
| CS-319    |
| CS-347    |
| FIN-201   |
| HIS-351   |
| MU-199    |
| PHY-101   |

**Set-Intersection Operation:**

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: r ∩ s
- **Assume:**  
r, s have the same arity  
attributes of r and s are compatible
- **Example:** Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\Pi_{course\_id} (\sigma_{semester="Fall" \wedge year=2017} (section)) \cap \Pi_{course\_id} (\sigma_{semester="Spring" \wedge year=2018} (section))$$

- **Result**

| course_id |
|-----------|
| CS-101    |

**Set Difference (Minus) Operation:**

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- **Notation:**  $r - s$
- Set differences must be taken between compatible relations.  
 $r$  and  $s$  must have the same arity  
 attribute domains of  $r$  and  $s$  must be compatible
- **Example:** to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester
- **Result:**

$\Pi_{course\_id}(course\_id, semester='Fall', Year=2017(section)) - \Pi_{course\_id}(course\_id, semester='Spring', Year=2018(section))$

| course_id |
|-----------|
| CS-347    |
| PHY-101   |

**Equivalent Queries**

- There is more than one way to write a query in relational algebra.
- **Example:** Find information about courses taught by instructors in the Physics department with salary greater than 90,000

Query 1

$\Pi_{dept\_name='Physics'}(\sigma_{salary > 90,000}(instructor))$

Query 2

$\sigma_{dept\_name='Physics'}(\sigma_{salary > 90,000}(instructor))$

The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

**3.1.2 The CARTESIAN PRODUCT (CROSS PRODUCT)**

**Operation Cartesian Product Operation in Relational Algebra**

- Applying CARTESIAN PRODUCT on two relations that is on two sets of tuples, it will take every tuple one by one from the left set(relation) and will pair it up with all the tuples in the right set(relation).
- So, the CROSS PRODUCT of two relation  $A(R_1, R_2, R_3, \dots, R_p)$  with degree  $p$ , and  $B(S_1, S_2, S_3, \dots, S_n)$  with degree  $n$ , is a relation  $C(R_1, R_2, R_3, \dots, R_p, S_1, S_2, S_3, \dots, S_n)$  with degree  $p + n$  attributes.
- **Notation:**  $A \times S$   
 where  $A$  and  $S$  are the relations,  
 the symbol ‘ $\times$ ’ is used to denote the CROSS PRODUCT operator.
- **Example:**  
 Consider two relations STUDENT(SNO, FNAME, LNAME) and  
 DETAIL(ROLLNO, AGE) below:

| SNO | FNAME  | LNAME  |
|-----|--------|--------|
| 1   | Albert | Singh  |
| 2   | Nora   | Fatehi |

| ROLLNO | AGE |
|--------|-----|
| 5      | 18  |
| 9      | 21  |

- On applying CROSS PRODUCT on STUDENT and  
DETAIL: STUDENT × DETAILS

| SNO | FNAME  | LNAME  | ROLLNO | AGE |
|-----|--------|--------|--------|-----|
| 1   | Albert | Singh  | 5      | 18  |
| 1   | Albert | Singh  | 9      | 21  |
| 2   | Nora   | Fatehi | 5      | 18  |
| 2   | Nora   | Fatehi | 9      | 21  |

So the number of tuples in the resulting relation on performing CROSS PRODUCT is  $2 \times 2 = 4$ .

The CARTESIAN PRODUCT operation also known as CROSS PRODUCT or CROSS JOIN denoted by  $\times$  is 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  $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$  is a relation Q with degree  $n+m$  attributes  $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , in that order. The resulting relation Q has one tuple for each combination of tuples one from R and one from S. Hence, if R has  $nR$  tuples (denoted as  $|R| = nR$ ), and S has  $nS$  tuples, then  $R \times S$  will have  $nR * nS$  tuples

**Example:** Suppose that we want to retrieve a list of names of each employee’s dependents.

```

FEMALE_EMPS ← σgender='F'(EMPLOYEE)
EMPNAMES ← πFname, Lname, Ssn(FEMALE_EMPS)
EMP_DEPENDENTS ← EMPNAMES × DEPENDENT
ACTUAL_DEPENDENTS ← σSsn=Essn(EMP_DEPENDENTS)
RESULT ← πFname, Lname, Dependent_name (ACTUAL_DEPENDENTS)
    
```

**FEMALE\_EMPS**

| Fname    | Minit | Lname   | Ssn       | Bdate      | Address                | gen | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|------------|------------------------|-----|--------|-----------|-----|
| Alicia   | J     | Zelaya  | 999887777 | 1968-07-19 | 3321Castle, Spring, TX | F   | 25000  | 987854321 | 4   |
| Jennifer | S     | Wallace | 987654321 | 1941-06-20 | 291Berry, Bellaire, TX | F   | 43000  | 888665555 | 4   |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX | F   | 25000  | 333445555 | 5   |

**EMPNAMES**

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

**RESULT**

| Fname    | Lname   | Dependent_name |
|----------|---------|----------------|
| Jennifer | Wallace | Abner          |

**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 | ... |

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT related tuples only from the two relations by specifying an appropriate selection condition after the Cartesian product.

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables.

**Cartesian Product Operation in Relational Algebra**

- Applying CARTESIAN PRODUCT on two relations that is on two sets of tuples, it will take every tuple one by one from the left set (relation) and will pair it up with all the tuples in the right set (relation).
- So, the CROSS PRODUCT of two relation A(R1, R2, R3, ..., Rp) with degree p, and B(S1, S2, S3, ..., Sn) with degree n, is a relation C(R1, R2, R3, ..., Rp, S1, S2, S3, ..., Sn) with degree p + n attributes.
- **Notation:**  $A \times S$   
where A and S are the relations,  
the symbol ‘ $\times$ ’ is used to denote the CROSS PRODUCT operator.
- **Example:**  
Consider two relations STUDENT(SNO, FNAME, LNAME) and  
DETAIL(ROLLNO, AGE) below:

| SNO | FNAME  | LNAME  | ROLLNO | AGE |
|-----|--------|--------|--------|-----|
| 1   | Albert | Singh  | 5      | 18  |
| 2   | Nora   | Fatehi | 9      | 21  |

On applying CROSS PRODUCT on STUDENT and DETAIL:  
STUDENT × DETAILS

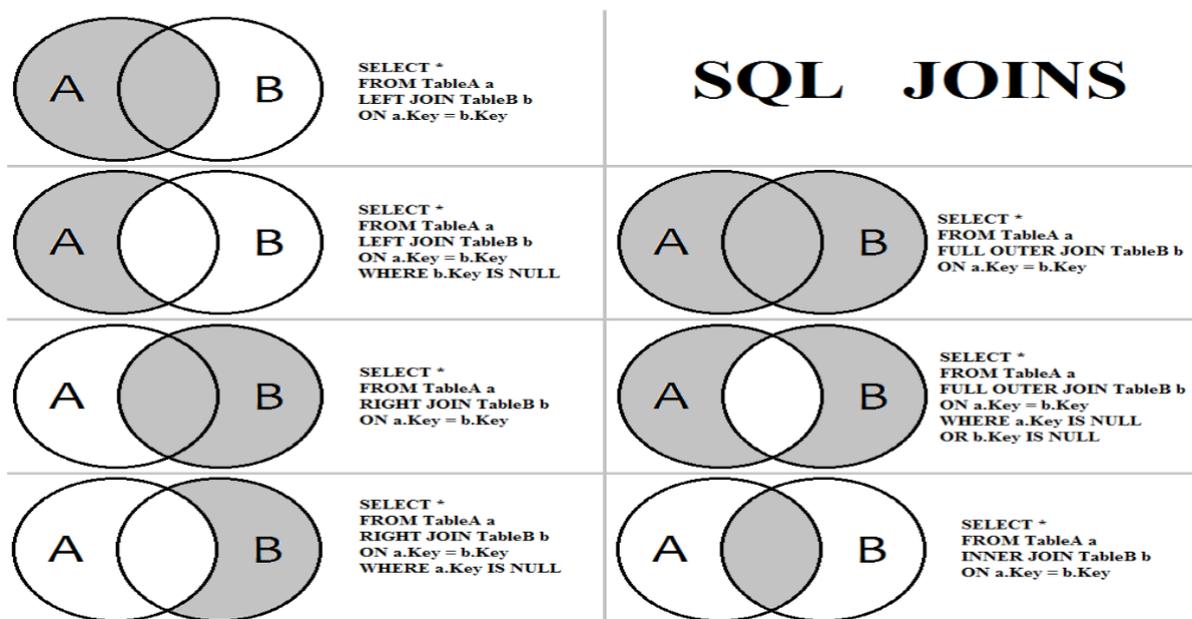
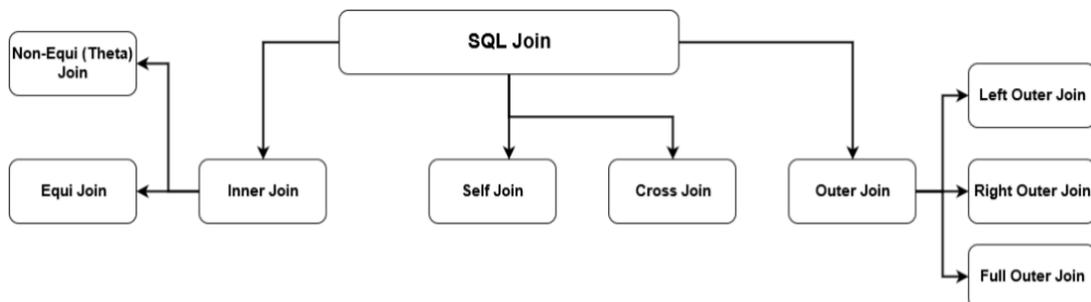
| SNO | FNAME  | LNAME  | ROLLNO | AGE |
|-----|--------|--------|--------|-----|
| 1   | Albert | Singh  | 5      | 18  |
| 1   | Albert | Singh  | 9      | 21  |
| 2   | Nora   | Fatehi | 5      | 18  |
| 2   | Nora   | Fatehi | 9      | 21  |

So the number of tuples in the resulting relation on performing CROSS PRODUCT is  $2 \times 2 = 4$ .

### 3.5 Binary Relational Operations: JOIN and DIVISION:

#### 3.5.1 The JOIN Operation

- **JOIN Operation:** Join in DBMS is a binary operation which allows you to combine join product and selection in one single statement.
- The goal of creating a join condition is that it helps you to combine the data from two or more DBMS tables.
- The tables in DBMS are associated using the primary key and foreign keys.
- **Types of Join**



**1. Natural Join:**

A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names. It is denoted by  $\bowtie$ .

**Example:** Let's use the EMPLOYEE table and SALARY table:

Input: {} EMP\_NAME, SALARY (EMPLOYEE  $\bowtie$  SALARY)

**Output:**

| EMP_NAME | SALARY |
|----------|--------|
| Stephan  | 50000  |
| Jack     | 30000  |
| Harry    | 25000  |

**2. Outer Join:**

The outer join operation is an extension of the join operation. It is used to deal with missing information.

**Example:**

EMPLOYEE

FACT\_WORKERS

| EMP_NAME | STREET      | CITY      |
|----------|-------------|-----------|
| Ram      | Civil line  | Mumbai    |
| Shyam    | Park street | Kolkata   |
| Ravi     | M.G. Street | Delhi     |
| Hari     | Nehru nagar | Hyderabad |

| EMP_NAME | BRANCH  | SALARY |
|----------|---------|--------|
| Ram      | Infosys | 10000  |
| Shyam    | Wipro   | 20000  |
| Kuber    | HCL     | 30000  |
| Hari     | TCS     | 50000  |

**Input:** (EMPLOYEE  $\bowtie$  FACT\_WORKERS)

**Output:**

| EMP_NAME | STREET      | CITY      | BRANCH  | SALARY |
|----------|-------------|-----------|---------|--------|
| Ram      | Civil line  | Mumbai    | Infosys | 10000  |
| Shyam    | Park street | Kolkata   | Wipro   | 20000  |
| Hari     | Nehru nagar | Hyderabad | TCS     | 50000  |

An outer join is basically of three types:

- a) Left outer join
- b) Right outer join
- c) Full outer join

a) **Left Outer Join:** Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

- In the left outer join, tuples in R have no matching tuples in S.

It is denoted by  $\ltimes$ .

- **Example:** Using the above EMPLOYEE table and FACT\_WORKERS table.

Input: EMPLOYEE  $\ltimes$  FACT\_WORKERS

- **Output:**

| EMP_NAME | STREET       | CITY      | BRANCH  | SALARY |
|----------|--------------|-----------|---------|--------|
| Ram      | Civil line   | Mumbai    | Infosys | 10000  |
| Shyam    | Park street  | Kolkata   | Wipro   | 20000  |
| Hari     | Nehru street | Hyderabad | TCS     | 50000  |
| Ravi     | M.G. Street  | Delhi     | NULL    | NULL   |

b) **Right outer join:** Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

- In right outer join, tuples in S have no matching tuples in R.
- **Example:** Using the above EMPLOYEE table and FACT\_WORKERS Relation  
Input: EMPLOYEE ⋈ FACT\_WORKERS
- **Output:**

| EMP_NAME | BRANCH  | SALARY | STREET       | CITY      |
|----------|---------|--------|--------------|-----------|
| Ram      | Infosys | 10000  | Civil line   | Mumbai    |
| Shyam    | Wipro   | 20000  | Park street  | Kolkata   |
| Hari     | TCS     | 50000  | Nehru street | Hyderabad |
| Kuber    | HCL     | 30000  | NULL         | NULL      |

c) **Full outer join:** Full outer join is like a left or right join except that it contains all rows from both tables.

- In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.
- **Example:** Using the above EMPLOYEE table and FACT\_WORKERS table  
Input: EMPLOYEE ⋈ FACT\_WORKERS
- **Output:**

| EMP_NAME | STREET       | CITY      | BRANCH  | SALARY |
|----------|--------------|-----------|---------|--------|
| Ram      | Civil line   | Mumbai    | Infosys | 10000  |
| Shyam    | Park street  | Kolkata   | Wipro   | 20000  |
| Hari     | Nehru street | Hyderabad | TCS     | 50000  |
| Ravi     | M.G. Street  | Delhi     | NULL    | NULL   |
| Kuber    | NULL         | NULL      | HCL     | 30000  |

3) **Equi join:** It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator(=).

**Example:**

CUSTOMER RELATION

| CLASS_ID | NAME    |
|----------|---------|
| 1        | John    |
| 2        | Harry   |
| 3        | Jackson |

PRODUCT

| PRODUCT_ID | CITY   |
|------------|--------|
| 1          | Delhi  |
| 2          | Mumbai |
| 3          | Noida  |

**Input:** CUSTOMER  $\bowtie$  PRODUCT

**Output:**

| CLASS_ID | NAME  | PRODUCT_ID | CITY   |
|----------|-------|------------|--------|
| 1        | John  | 1          | Delhi  |
| 2        | Harry | 2          | Mumbai |
| 3        | Harry | 3          | Noida  |

The JOIN operation, denoted by  $\bowtie$ , is used to combine related tuples from two relations into single longer tuples. It allows us to process relationships among relations. The general form of a JOIN operation on two relations R(A1, A2, ..., An) and S(B1, B2, ..., Bm) is

$$R \bowtie \langle \text{join condition} \rangle S$$

**Example:** 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

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

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 | ... |

The result of the JOIN is a relation Q with n + m attributes Q(A1, A2, ..., An, B1, B2, ..., Bm) in that order. Q has one tuple for each combination of tuples one from R and one from S whenever the combination satisfies the join condition. This is the main difference between CARTESIAN PRODUCT and JOIN. In JOIN, only combinations of tuples satisfying the join condition appear in the result, whereas in the CARTESIAN PRODUCT all combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples.

Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q as a single combined tuple. A general join condition is of the form

$$\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$$

where each <condition> is of the form  $A_i B_j$ ,  $A_i$  is an attribute of R,  $B_j$  is an attribute of S,  $A_i$  and  $B_j$  have the same domain, and ( $\theta$ ) is one of the comparison operators  $\{=, <, >, \leq, \geq, \neq\}$ . A JOIN operation with such a general join condition is called as **THETA JOIN**. Tuples whose join attributes are NULL or for which the join condition is FALSE do not appear in the result.

**3.5.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN**

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an EQUIJOIN. In the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple.

For example 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.

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. 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 \* (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.

**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     |

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

**DEPT\_LOCS ← DEPARTMENT \* DEPT\_LOCATIONS**

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   |

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. If no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples.

A more general, but nonstandard definition for NATURAL JOIN is

$$Q \leftarrow R \bowtie_{(\langle \text{list1} \rangle) = (\langle \text{list2} \rangle)} S$$

where,

<list1> : list of i attributes from R,

<list2> : list of i attributes from S

The lists are used to form equality comparison conditions between pairs of corresponding attributes and then the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R <list1> is kept in the result Q.

In general, if R has nR tuples and S has nS tuples, the result of a JOIN operation  $R \bowtie_{\langle \text{join condition} \rangle} S$  will have between zero and nR \* nS tuples. The expected size of the join result divided by the maximum size nR \* nS leads to a ratio called join selectivity, which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

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. Informally, an inner join is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. 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:

$$((PROJECT \bowtie_{Dname=Dnumber} DEPARTMENT) \bowtie_{Mgr\_ssn=Dmgr} EMPLOYEE)$$

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

In SQL, JOIN can be realized in several different ways

- The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions.
- The second way is to use a nested relation
- Another way is to use the concept of joined tables

**3.5.3 A Complete Set of Relational Algebra Operations**

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:

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

As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation,

$$R \bowtie_{\langle \text{condition} \rangle} S = \sigma_{\langle \text{condition} \rangle} (R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also not strictly necessary for the expressive power of the relational algebra.

**3.5.4 The DIVISION Operation**

- **Division operation:** The division operator is used for queries which involve the ‘all’.  $R1 \div R2 = \text{tuples of } R1 \text{ associated with all tuples of } R2$ .
- **Example:** Retrieve the name of the subject that is taught in all courses.

| Name     | Course |
|----------|--------|
| System   | Btech  |
| Database | Mtech  |
| Database | Btech  |
| Algebra  | Btech  |

+

| Course |
|--------|
| Btech  |
| Btech  |

=

| Name     |
|----------|
| database |

- The expression:

Smith  $\leftarrow \Pi_{Pno}(\sigma_{Ename = 'john smith'}(employee * works\ on\ Pno=Eno))$

Consider the Employee table given below –

| Name  | Eno | Pno |
|-------|-----|-----|
| John  | 123 | P1  |
| Smith | 123 | P2  |
| A     | 121 | P3  |

+

Works on the following –

| Eno | Pno | Pname  |
|-----|-----|--------|
| 123 | P1  | Market |
| 123 | P2  | Sales  |

=

The result is as follows

| Eno |
|-----|
| 123 |

The DIVISION operation, denoted by  $\div$ , is useful for a special kind of query that sometimes occurs in database applications. An example is Retrieve the names of employees who work on all the projects that ‘John Smith’ works on. To express this query using the DIVISION operation, proceed as follows.

- First, retrieve the list of project numbers that ‘John Smith’ works on in the intermediate relation SMITH\_PNOS:

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

- Next, create a relation that includes a tuple  $\langle Pno, Essn \rangle$  whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN\_PNOS:

$SMITH \leftarrow \sigma_{Fname='John' \text{ AND } Lname='Smith'}(EMPLOYEE)$   
 $SMITH\_PNOS \leftarrow \pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn} SMITH)$

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

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

(a)

| SSN_PNOS  |     |
|-----------|-----|
| Essn      | Pno |
| 123456789 | 1   |
| 123456789 | 2   |
| 666684444 | 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  |

| SSNS      |
|-----------|
| Ssn       |
| 123456789 |
| 453453453 |

| SMITH_PNOS |
|------------|
| Pno        |
| 1          |
| 2          |

In general, the DIVISION operation is applied to two relations  $R(Z) \div S(X)$ , where the attributes of R are a subset of the attributes of S; that is,  $X \subseteq Z$ . Let Y be the set of attributes of R that are not attributes of S; that is,  $Y = Z - X$  (and hence  $Z = X \cup Y$ ). The result of DIVISION is a relation  $T(Y)$  that includes a tuple t if tuples tR appear in R with  $tR[Y] = t$ , and with  $tR[X] = tS$  for every tuple tS in S. This means that, for a tuple t to appear in the result T of t

Figure below illustrates a DIVISION operation where  $X = \{A\}$ ,  $Y = \{B\}$ , and  $Z = \{A, B\}$ .

| 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 |

The tuples (values) b1 and b4 appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: b2 does not appear with a2, and b3 does not appear with a1.

The DIVISION operation can be expressed as a sequence of  $\pi$ ,  $\times$  and  $-$  operations are as follows:

$$\begin{array}{l}
 T1 \leftarrow \pi_Y(R) \\
 T2 \leftarrow \pi_Y((S \times T1) - R) \\
 T \leftarrow T1 - T2
 \end{array}$$

| OPERATION         | PURPOSE   | NOTATION   |
|-------------------|---|--|
| SELECT            | Selects all tuples that satisfy the selection condition from a relation $R$ .   | $\sigma_{\langle \text{selection condition} \rangle}(R)$   |
| PROJECT           | Produces a new relation with only some of the attributes of $R$ , and removes duplicate tuples.   | $\pi_{\langle \text{attribute list} \rangle}(R)$   |
| THETA JOIN        | Produces all combinations of tuples from $R_1$ and $R_2$ that satisfy the join condition.   | $R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$  |
| EQUJOIN           | Produces all the combinations of tuples from $R_1$ and $R_2$ that satisfy a join condition with only equality comparisons.  | $R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$ OR<br>$R_1 \bowtie_{\langle \text{join attributes } \{ \rightarrow \} \rangle} R_2$<br>$\quad \quad \quad \langle \text{join attributes } \{ \rightarrow \} \rangle R_2$                   |
| NATURAL JOIN      | Same as EQUJOIN 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. | $R_1 \star_{\langle \text{join condition} \rangle} R_2$<br>OR $R_1 \star_{\langle \text{join attributes } \{ \rightarrow \} \rangle} R_2$<br>$\quad \quad \quad \langle \text{join attributes } \{ \rightarrow \} \rangle R_2$<br>OR $R_1 \star R_2$ |
| 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 $Z = X \cup Y$ .                          | $R_1(Z) \div R_2(Y)$   |

Table: Operations of Relational Algebra

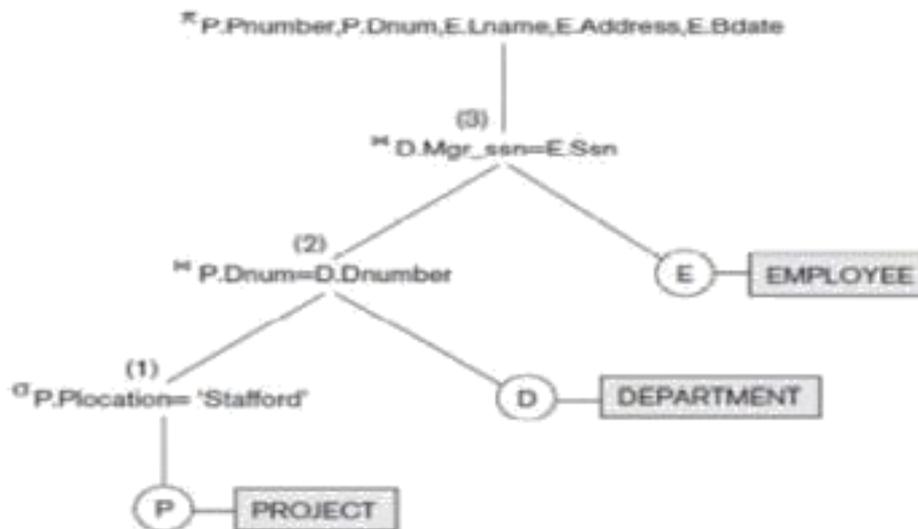
3.5.5 Notation for Query Trees

Query tree (query evaluation tree or query execution tree) is used in relational systems to represent queries internally. 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.

**Example:** 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.

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



Leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE. The relational algebra operations in the expression are represented by internal tree nodes. The query tree signifies an explicit order of execution in the following sense. The node marked (1) 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.

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.

### 3.7 Examples of Queries in Relational Algebra:

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

```

RESEARCH_DEPT ← σDname='Research'(DEPARTMENT)
RESEARCH_EMPS ← (RESEARCH_DEPT ⋈Dnumber=Dno EMPLOYEE)
RESULT ← πFname, Lname, Address(RESEARCH_EMPS)

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

πFname, Lname, Address (σDname='Research'(DEPARTMENT ⋈Dnumber=Dno(EMPLOYEE)))
    
```

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

```

STAFFORD_PROJS ← σPlocation='Stafford'(PROJECT)
CONTR_DEPTS ← (STAFFORD_PROJS ⋈Dnum=Dnumber DEPARTMENT)
PROJ_DEPT_MGRS ← (CONTR_DEPTS ⋈Mgr_ssn=Ssn EMPLOYEE)
RESULT ← πPnumber, Dnum, Lname, Address, Bdate(PROJ_DEPT_MGRS)
    
```

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

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

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

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

Query 5. List the names of all employees with two or more dependents.

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

Query 6. Retrieve the names of employees who have no dependents.

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

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

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