Database Management System (MDS 505)

Jagdish Bhatta

Unit-2

The Relational Languages and Relational Model: SQL

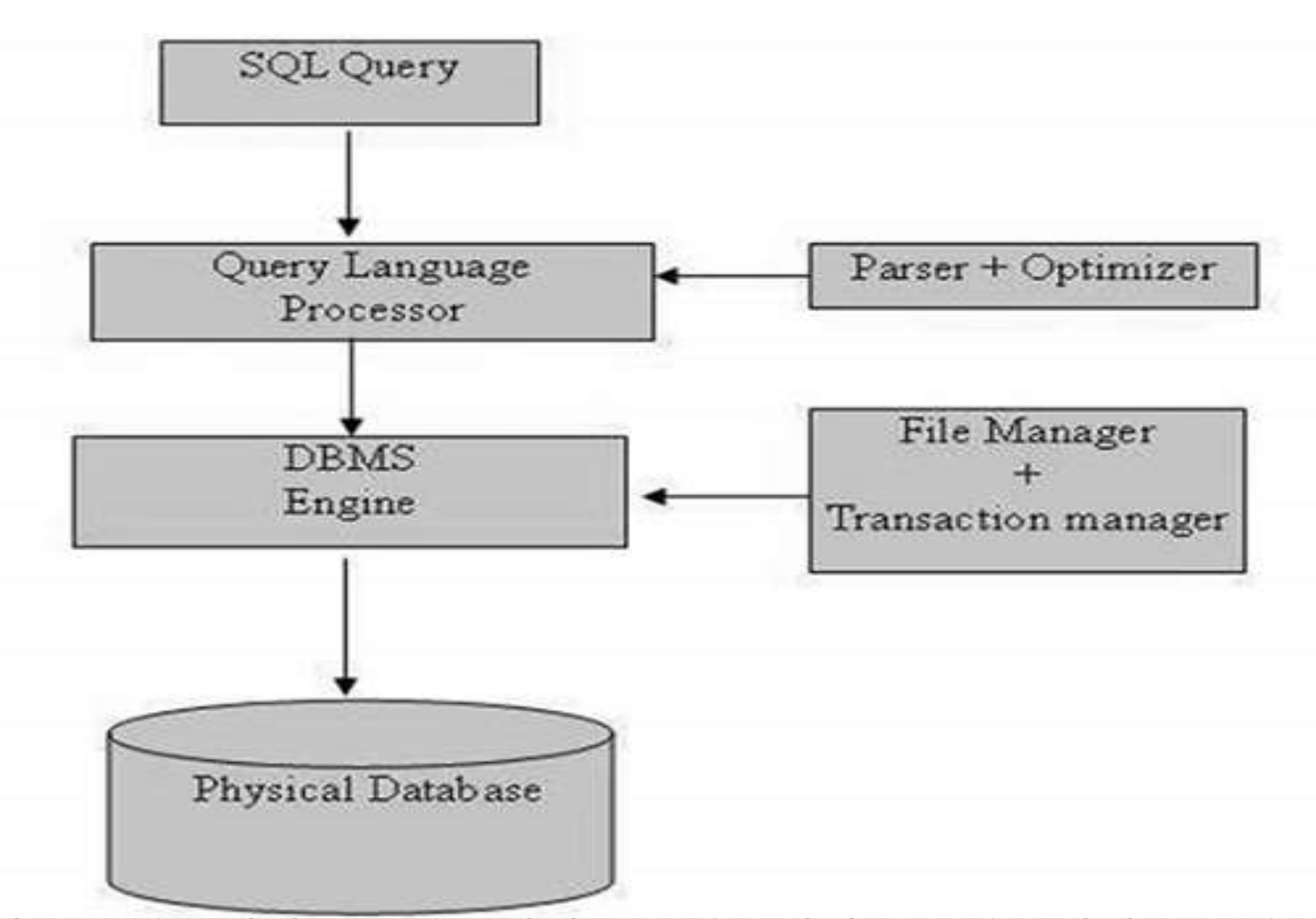
Introduction

- Originally, Structured Query Language (SQL), was called SEQUEL (for Structured English Query Language) and was designed & implemented at IBM Research.
- SQL is the most popular and most user friendly query language. SQL uses a combination of *relational-algebra* and *relational-calculus* constructs.
- Although we refer to the SQL language as a "*query language*", it can be used for defining the structure of the data, modifying the data in the database, and specifying security constraints.
- SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Introduction

- SQL has the following features:
  - **Data-definition language(DDL): -** The SQL DDL provides commands for defining relation schemas and modifying relation schemas.
  - **Interactive data-manipulation language(DML): -** The SQL DML includes a query language based on both the relational algebra and tuple relational calculus. It also includes commands to insert, delete, and modify tuples.
  - **View definition: -** The SQL DDL includes commands for defining views.
  - **Transaction control: -** SQL includes commands for specifying the beginning and ending of transactions.
  - **Embedded SQL and dynamic SQL: -** Embedded SQL and dynamic SQL defines how SQL statements can be embedded within general purpose programming languages, such as PHP, Java etc.
  - **Integrity: -** The SQL DDL includes commands for specifying integrity constraints.
  - **Authorization: -** The SQL DDL includes commands for specifying access rights to relations and views.

SQL Process


SQL Query
Query Language Processor
Parser + Optimizer
DBMS Engine
File Manager + Transaction manager
Physical Database

SQL Commands

- **DDL - Data Definition Language**
  - **CREATE:** Creates a database, new table, a view of a table, or other object in the database.
  - **ALTER:** Modifies an existing database object, such as a table.
  - **DROP:** Deletes an entire table, a view of a table or other objects in the database.
- **DML - Data Manipulation Language**
  - **SELECT:** Retrieves certain records from one or more tables.
  - **INSERT:** Creates a record.
  - **UPDATE:** Modifies records.
  - **DELETE:** Deletes records.

SQL Commands

- **DCL - Data Control Language**
  - **GRANT:** Gives a privilege to user.
  - **REVOKE:** Takes back privileges granted from user.

Basic Data Types in SQL

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(*i*, *j*)—or DEC(*i*, *j*) or NUMERIC(*i*, *j*)—where *i*, the *precision*, is the total number of decimal digits and *j*, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

Basic Data Types in SQL

- **Character-string** data types are either fixed length—CHAR(*n*) or
- CHARACTER(*n*), where *n* is the number of characters—or varying length— VARCHAR(*n*) or CHAR VARYING(*n*) or CHARACTER VARYING(*n*), where *n* is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive* (a distinction is made between uppercase and lowercase). For fixed length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith' if needed. Padded blanks are generally ignored when strings are compared.

Basic Data Types in SQL

- **Character-string** data types
- 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). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

Basic Data Types in SQL

- **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; for example, B‘10101’.5 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. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.

Basic Data Types in SQL

- **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.
- The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.

Null Value

- Each type may include a special value called the **null value. A null value** indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered, as we shall see shortly.

Specifying Constraints

- Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
- Constraints can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.

SQL Constraints

- **Not Null:** It disallows NULL as a valid value. Ensures a column in a table can not be null.

  (SQL allows the use of null values to indicate that values either unknown or does not exist.)
- **Default <Value> :** It specifies default value of an attribute. Without default clause, the default value is null for those attributes without having Not Null constraint.
- **Primary Key:** It specifies one or more attributes as a primary key and uniquely identifies each row/record in a database table.
- **Foreign Key:** It ensures referential integrity. A referential integrity constraint can be violated when tuples are inserted or deleted or when a primary or foreign key attribute value is modified. The default action for integrity violation is to reject the update operation that leads to violation. However one can specify a referential triggered action clause to any foreign key constraint. The option include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE.

SQL Constraints

- **Check:** It allows to specify a predicate that must be satisfied by any value assigned to an attribute.

  E.g. Dnumber INT Not Null Check (Dnumber>0)

- **Index:** Used to create and retrieve data from the database very quickly.
- **Unique:** Ensures that all the values in a column are different. The **unique specification says that attributes Aj1 , Aj2, . . . , Ajm form a candidate** key; that is, no two tuples in the relation can be equal on all the listed attributes. However, candidate key attributes are permitted to be null unless they have explicitly been declared to be **not null.**

  E.g. **unique (*Aj1 , Aj2, . . . , Ajm)***

SQL Constraints

- **Constraints for Referential Actions**
  - CASCADE: Delete or update the row from the parent table, and automatically delete or update the matching rows in the child table. Both ON DELETE CASCADE and ON UPDATE CASCADE are supported.
  - SET NULL: Delete or update the row from the parent table, and set the foreign key column or columns in the child table to NULL.
  - RESTRICT: Rejects the delete or update operation for the parent table.
  - NO ACTION: Rejects the delete or update operation for the parent table if there is a related foreign key value in the referenced table. A keyword from standard SQL. In MySQL, equivalent to RESTRICT.
  - SET DEFAULT: sets the referenced value to some default on the delete or update operation.

SQL Data Definition

- The set of relations in a database must be specified to the system by means of a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:
  - The schema for each relation.
  - The types of values associated with each attribute.
  - The integrity constraints.
  - The set of indices to be maintained for each relation.
  - The security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Basic SQL DDL Commands

- **For Creating Database Schema:**

  - CREATE SCHEMA <database-name>AUTHORIZATION <user-identifier>;

  For Example:- Create Schema Employee Authorization Jagdish;

- **For Creating Tables:** Create table command is used to specify a new relation by giving its name and specifying its attributes , their data types and initial constraints, if any. The syntax is as follows:
  - CREATE TABLE schema].table_name [{column descriptions}];
  - CREATE TABLE table_name [{column descriptions}];

- For Creating Domains: Create domain command is used to create a domain of particular data type. The syntax is;
  - CREATE DOMAIN domain_name AS data type

Basic SQL DDL Commands

- **For Databases:**
- The database is created using create database statement.

  For Example:- CREATE DATABASE COMPANY;

- **Using the Databases:**
- The database is used before creating or accessing tables within it.

  For Example:- USE COMPANY;

- **For Domains:**
- The domain is created using create domain statement.

  For Example:- **CREATE DOMAIN** SSN_TYPE **AS** CHAR(9);
- We can use SSN_TYPE in place of CHAR(9). (This feature may not be available in some implementations of SQL)

Basic SQL DDL Commands

- **For Tables:**
- The table created through create table statements are called base tables (base relations) means that the relation & its tuples are actually created & stored as a file by DBMS. The tables created by VIEW are virtual tables and may not correspond to any physical table

For Example:-

```
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));
```

Basic SQL DDL Commands: Referencing

For Example:- CREATE TABLE Employee

```
(       SSN         varchar(10) NOT NULL,
        Fname       varchar(20) NOT NULL,
        Lname       varchar(20) NOT NULL,
        Bdate       date,
        Address     varchar(30),
        Sex         char,
        Salary      decimal(10,2),
        SuperSSN    varchar(10),
        Dno         INT         NOT NULL,
        PRIMARY KEY (SSN),
        FOREIGN KEY (SuperSSN) REFERENCES Employee(SSN),
        FOREIGN KEY (Dno) REFERENCES Department(Dnumber)
);
```

Basic SQL DDL Commands

```
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) );
```

Basic SQL DDL Commands

```
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) );
```

Basic SQL DDL Commands: Specifying Constraints

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

Basic SQL DDL Commands: Specifying Constraints

- **Using if not exists in create query:**

```
CREATE TABLE IF NOT EXISTS newbook
( book_id varchar(15) NOT NULL UNIQUE,
book_name varchar(50) ,
isbn_no varchar(15) NOT NULL UNIQUE ,
cate_id varchar(8) ,
aut_id varchar(8) ,
pub_id varchar(8) ,
book_price decimal(8,2) ,
PRIMARY KEY (book_id) );
```

Basic SQL DDL Commands: Specifying Constraints

- **Using if not exists in create query and check constraint:**

```
CREATE TABLE IF NOT EXISTS parts (
    part_no VARCHAR(18) PRIMARY KEY,
    description VARCHAR(40),
    cost DECIMAL(10 , 2 ) NOT NULL CHECK(cost > 0),
    price DECIMAL (10,2) NOT NULL
);
```

- **Using auto increment:**

```
CREATE TABLE IF NOT EXISTS newauthor (
id int NOT NULL AUTO_INCREMENT,
aut_id varchar(8),
aut_name varchar(50),
country varchar(25),
home_city varchar(25) NOT NULL,
PRIMARY KEY (id) );
```

Basic SQL DDL Commands: Specifying Constraints

- **Multiple Primary keys:**

```
CREATE TABLE IF NOT EXISTS newauthor(
aut_id varchar(8) NOT NULL ,
aut_name varchar(50) NOT NULL,
country varchar(25) NOT NULL,
home_city varchar(25) NOT NULL,
PRIMARY KEY (aut_id, home_city));
```

- **Using CHECK constraint using IN:**

```
CREATE TABLE IF NOT EXISTS newauthor(
aut_id varchar(8) NOT NULL ,
aut_name varchar(50) NOT NULL,
country varchar(25) NOT NULL CHECK (country IN ('USA','UK',Nepal')),
home_city varchar(25) NOT NULL,
PRIMARY KEY (aut_id,home_city));
```

Basic SQL DDL Commands: Specifying Constraints

- **Using CHECK constraint with LIKE:**

```
CREATE TABLE IF NOT EXISTS newbook
( book_id varchar(15) NOT NULL UNIQUE,
book_name varchar(50) ,
isbn_no varchar(15) NOT NULL UNIQUE ,
cate_id varchar(8) ,
aut_id varchar(8) ,
pub_id varchar(8) ,
dt_of_pub date CHECK (dt_of_pub LIKE '--/--/----'),
pub_lang varchar(15) ,
no_page decimal(5,0) CHECK(no_page>0) ,
book_price decimal(8,2) ,
PRIMARY KEY (book_id) );
```

Statements for Changing the schema

- **Drop Command:** Drop command can be used to drop the schema elements as tables, domains, or constraints. One can also drop a schema itself.

- **Schema/Database Deletion:**

  DROP SCHEMA Schema_name [CASCADE / RESTRICT]

  DROP DATABASE [IF EXISTS] { database_name}

  This statement drops the schema. If cascade is used, then all tables, domains, and other elements are also deleted along with the schema. While if restrict option is used, then the schema is dropped only if it has no elements in it.

  - E.g. DROP SCHEMA Company CASCADE;
  - DROP DATABASE Company;
  - DROP DATABASE IF EXISTS Company;

Statements for Changing the schema

- **Drop Command:** Drop command can be used to drop the schema elements as tables, domains, or constraints. One can also drop a schema itself.

- **Table Deletion:**

  DROP TABLE Table_name [CASCADE / RESTRICT]

  This statement drops the table. If restrict is used, then the table is dropped only if it is not referenced in any constraint. While if Cascade option is used, all the constraints & views that references the table are dropped automatically from the schema along with the table itself.

  - E.g. DROP TABLE Employee CASCADE;

The Alter Command

- The alter command is used to change the definition of the base table and other schema elements. For base tables, the possible alter table actions include adding or dropping a column(attribute), changing a column definition, adding or dropping table constraints.
  - **To add an attribute:** *ALTER TABLE Table_name ADD [Column_name]*

    E.g.: *ALTER TABLE Employee ADD Jobtype varchar(30);*
  - **To drop an attribute/column:** To drop a column, we can use CASCADE or RESTRICT option. With Cascasde, all constraints, views that reference the column are dropped automatically from the schema, along with the column. If restrict is chosn, the command is successful only if no views or constraints reference the column. The Syxtanx is:

    ALTER TABLE Table_name DROP [Column_name] [Cascade/Restrict]

    E.g.: *ALTER TABLE Employee DROP Address CASCADE;*
  - **To set or drop default value of an attribute:**

    E.g.: ALTER TABLE Department ALTER MgrSSN DROP DEFAULT;

    ALTER TABLE Department ALTER MgrSSN SET DEFAULT "111";

The Alter Command

– **To Add & Drop constraint:**

E.g.: CREATE TABLE Dependent
(
ESSN varchar(10),
Dependent_name varchar(30) CONSTRAINT Name_Not_Null NOTNULL,
Sex char,
Relationship varchar(10) CONSTRAINT Rel_None DEFAULT 'none',
CONSTRAINT DEPENDENT_PK PRIMARY KEY (ESSN, Dependent_name)
);

Now to drop the constraint:

ALTER TABLE Dependent DROP CONSTRAINT Name_Not_Null ;

To add new constraint:

ALTER TABLE Dependent ADD CONSTRAINT DEP_FK FOREIGN KEY (ESSN)
REFERENCES Employee(SSN);

The Alter Command

- **To Add & Drop index:**

  ALTER TABLE *table_name* ADD INDEX *index_name* (column_name);

  ALTER TABLE *table_name* DROP INDEX *index_name*;

The Drop Command

- The DROP command can be used to drop *named schema elements*, *such as tables*, domains, types, or constraints. One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two *drop behavior* options: CASCADE and RESTRICT.
- For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

  DROP SCHEMA COMPANY CASCADE;

  DROP DATABASE COMPANY;

- If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements in it; otherwise, the DROP command will not be executed. To* use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

The Drop Command

- If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, we can perform;

  DROP TABLE DEPENDENT CASCADE;

- If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced in any constraints (for example, by foreign key definitions in another* relation) or views or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.
- Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition*.
- If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command.

The Truncate Command

- The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.
- E.g.: TRUNCATE TABLE Employee;
- You need not have to specify a WHERE clause in a TRUNCATE TABLE statement.
- TRUNCATE TABLE cannot used when a foreign key references the table to be truncated, since TRUNCATE statements do not fire triggers. This could result in inconsistent data because ON DELETE/UPDATE triggers would not fire.

Basic Retrieval Queries in SQL

DML Statements in SQL

- **Select Statement:**
- SQL has one basic statement for retrieving information from a database; the SELECT statement.
- The basic form of Select statement is formed three clauses SELECT, FROM, and WHERE and has following structure:

  SELECT <Attribute List>

  FROM <Table List>

  WHERE <Condition>

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

DML Statements in SQL

- **Select Statement:**
- The SELECT clause corresponds to project operation of the relational algebra. It is used to list attributes desired in the result of query.
- The FROM clause corresponds to Cartesian-product of the relational algebra. It lists the relations to be scanned in the evaluation of expression.
- The WHERE clause corresponds to the selection predicate of the relation algebra. It consist of a predicate involving attributes of the relations that appear from clause.
- Example:

```
SELECT SSN, Fname, Lname
FROM Employee
WHERE Address= ‘Kathmandu’;
```

DML Statements in SQL

- **Select Statement:**
- A typical select query has the form

  Select $A_1 \ldots\ldots A_n$

  From $r_1, r_2, r_3 \ldots\ldots\ldots\ldots r_m$

  Where P

  This query is equivalent to following relational algebra query;

  $\pi_{Ai \ldots An} (\sigma_P(r_1 \times r_2 \times r_3 \ldots\ldots\ldots \times r_m))$

- The result of the select query *may contain* duplicate tuples. To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used.

```
SELECT DISTINCT Fname
FROM Employee
WHERE Address= 'Kathmandu';
```

Ambiguous Attribute Name

- Some multi table queries may have same attribute name, so accessing them can be **ambiguous**. Thus to prevent this **ambiguity**, *prefixing* the relation name to the attribute name and separating the two by a period is done.
- Example: For each employee, retrieve the employee's name, and the name of his or her department.

  Consider the schemas are

  EMPLOYEE(Name, SSN, Address, Dob, Dno)

  DEPARTMENT(Name, Dnumber)

```
SELECT   Employee.Name, Department.Name
FROM     Employee, Department
WHERE    Employee.Dno=Department.Dnumber
```

- **Here both employee and department have same attribute name "Name" so this forms ambiguous attribute, which can be accessed as above.**

Aliasing in SQL

- Some queries need to refer to the same relation twice. In this case, *aliases* are given to the relation name.
- Example: For each employee, retrieve the employee's name, and the 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.SUPERSSN = S.SSN
```

  - In above query, the alternate relation names E and S are called *aliases* or *tuple variables* for the EMPLOYEE relation.
  - We can think of E and S as two *different copies* of EMPLOYEE; E represents employees in role of *supervisees* and S represents employees in role of *supervisors.*

Aliasing in SQL

- We can also rename the table names to shorter names by creating an *alias* for each table name to avoid repeated typing of long table names.
- Example: For each employee, retrieve the employee's names.

```
SELECT   E.Fname, E.Lname
FROM     Employee AS E
```

- In above query, rather than typing Employee again and again we can use the alias E, which is shorter.

Aliasing and Renaming of Attribute in SQL

- It is also possible to **alias attributes** in the result of a query.

```
SELECT   SSN as Employee_SSN
FROM     Employee
WHERE    Employee.Fname="John"
```

- It is also possible to **rename the relation and attributes** within the query in SQL by giving them aliases.

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

- Here, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on. The above query results all of records from Employee where the resulted records will have attribute names Fn, Mi, Ln, and so on.

Unspecified Where clause & Use of *

- A *missing WHERE-clause* indicates no condition; hence, *all tuples* of the relations in the FROM-clause are selected.
- Example: Retrieve the SSN values for all employees.

```
SELECT   SSN
FROM     Employee
```

- If more than one relation is specified in the FROM-clause *and* there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected.
- Example: Select all combinations of Employee SSN and Department Name.

```
SELECT SSN, Dname
FROM   Employee, Department
```

Unspecified Where clause & Use of *

- To retrieve all the attribute values of the selected tuples, a * is used, which stands for *all the attributes*.
- Example: Retrieve all the attribute values of any Employee who work in department number 5.

```
SELECT  *
FROM    Employee
WHERE   DNO=5
```

- Example: Retrieve all the attributes of Employee & Department in which every employee works for “Research” department.

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

Tables as Set 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.
- If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT **ALL** does not.
- **For Example:** Retrieve the salary of every employee.

  SELECT ALL Salary **FROM** EMPLOYEE;
- **For Example:** Retrieve the distinct salary values of employee.

  SELECT DISTINCT Salary **FROM** EMPLOYEE;

Set Operations

- SQL has directly incorporated some set operations. They are union operation (**UNION)**, set difference (**MINUS/EXCEPT)** and intersection (**INTERSECT)**. The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result.*
- The set operations apply only to *union compatible relations* ; the two relations must have the same attributes and the attributes must appear in the same order.
- Example:
  - SELECT DISTINCT Fname

    FROM Employee

    WHERE Salary > 300000.00

    UNION

    SELECT DISTINCT Fname

    FROM Employee

    WHERE Salary < 24000.0

Substring Comparision

- The most commonly used operation on strings is pattern matching using the operator LIKE. We describe patterns by using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Examples:
  - 'Perry%' matches any string beginning with "Perry".
  - '%Perry' matches any string ending with "Perry".
  - '%Perry%' matches any string containing "Perry" as a substring.
  - '---' matches any string of exactly three characters.
  - '---%' matches any string of at least three characters.
- Example: Retrieve all employee whose first name consist "Arun".

```
SELECT *
FROM   Employee
WHERE  Fname LIKE '%Arun%';
```

Arithmetic Operators

- Standard numeric operators like addition (+), subtraction (-), multiplication (*) and division (/) can be applied to attributes having numeric domains.

- Example: Retrieve the resulting salary if every employee having salary greater than 30000 is given a 10 percent raise in the salary.

```
SELECT   1.1 * SALARY
FROM     EMPLOYEE
WHERE    SALARY > 30000;
```

Between Comparison Operators

- Between clause can be used as a comparison operator to check the value in between some values.
- 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)).

Ordering of Query Results

- The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s).
- The default order is in ascending order of values
- We can specify the keyword **DESC** if we want a descending order; the keyword **ASC** can be used to explicitly specify ascending order, even though it is the default
- Example: Retrieve names of all the employees in ascending order of their first name.

```
SELECT   Fname, Minit, Lname
FROM     Employee
ORDER BY Fname;
```

- Example: Retrieve names of all the employees in descending order of their first name.

```
SELECT   Fname, Minit, Lname
FROM     Employee
ORDER BY Fname DESC;
```

Ordering of Query Results

- Example: Retrieve names of all the employees, in alphabetical order, working for “Research” department.

```
SELECT    Fname, Lname
FROM      Employee, Department
WHERE     Dname='Research' AND Dno=Dnumber
ORDER BY Fname;
```

NULL Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes.
- *Null* signifies an unknown value or that a value does not exist.
- The predicate **IS NULL** or **IS NOT NULL** can be used to check for null values. *So equality comparison is not appropriate* .
- The result of any arithmetic expression involving *null* is *null*.
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.
- Example: Retrieve the names of all employees who do not have supervisors.

```
SELECT  Fname, Lname
FROM    Employee
WHERE   SuperSSN  IS  NULL
```

Nested Queries

- Nested queries are those in which with in the WHERE clause, there is a complete SELECT-FROM-WHERE statement.

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

```
SELECT   Fname, Lname, Address
FROM     Empyoee
WHERE    DNO IN  (SELECT  Dnumber
                  FROM    Department
                  WHERE  Dname='Research' )
```

- The outer query select an EMPLOYEE tuple if its DNO value is in the result of either nested query.

- The comparison operator **IN** compares a value v with a set (or multi-set) of values V, and evaluates to **TRUE** if v is one of the elements in V.

- In general, we can have several levels of nested queries.

Correlated Nested Queries

- A subquery that uses a correlation name from an outer query is called a **correlated subquery.**
- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query* , the two queries are said to be *correlated.*
- The result of a correlated nested query is *different for each tuple (or combination of tuples) of the relation(s) the outer query*
- Example: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
SELECT   E.Fname, E.Lname
FROM     Employee AS E
WHERE    E.SSN IN (SELECT     ESSN
                   FROM       Dependent
                   WHERE      ESSN=E.SSN AND
                              E.Fname=Dependent_name)
```

The EXISTS Function

- EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not.
- EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query.
- Example: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
SELECT   Fname, Lname
FROM     Employee AS E
WHERE    EXISTS (SELECT      *
                 FROM        Dependent
                 WHERE       E.SSN=ESSN AND
                             E.Fname=Dependent_name)
```

- Here for each *Employee tuple*, evaluate the nested query, which retrieves all *Dependent tuples* with the same employee number & name as the *Employee tuple*; if at least one tuple Exists in the result of the nested query, then select that *Employee tuple*.

The EXISTS Function

- In general, EXITS(Q) returns TRUE if there is at least one tuple exits in result of the nested query Q, & it returns FALSE otherwise. On the other hand, NOT EXISTS returns TRUE if there are no tuples in the result of nested query Q, and it returns FALSE otherwise.
- Example: Retrieve the names of employees who have no dependents.

```
SELECT      Fname, Lname
FROM        Empoyee
WHERE       NOT EXISTS   (SELECT *
                          FROM  Dependent
                          WHERE SSN=ESSN)
```

- In above query, the correlated nested query retrieves all *Dependent tuples* related to an *Employee tuple*. If *none exist* , the EMPLOYEE tuple is selected.

Explicit Sets

- It is also possible to use an **explicit (enumerated) set of values** in the WHERE-clause rather than a nested query
- Example: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

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

- Example: Retrieve the social security numbers of all employees who DOES NOT work on project number 1, 2, or 3.

```
SELECT   DISTINCT   ESSN
FROM    WORKS_ON
WHERE PNO NOT IN  (1, 2, 3);
```

Set Comparision

- SQL also allows < ***some,*** <= ***some,*** >= ***some,*** = ***some, and*** <> ***some comparisons.***
- **SQL also allows** < ***all,*** <= ***all,*** >= ***all,*** = ***all, and*** <> ***all*** comparisons.

Set Comparison

- As an example of the ability of a nested subquery to compare sets, consider the query **"Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."**

  SELECT distinct *T.name*

  FROM *instructor as T, instructor as S*

  WHERE *T.salary > S.salary and S.dept_name = 'Biology';*

- SQL does, however, offer an alternative style for writing the preceding query. The phrase "**greater than at least one**" is represented in SQL by **> *some.***

  SELECT *name*

  FROM *instructor*

  WHERE *salary > some (SELECT salary*

  FROM *instructor*

  WHERE *dept_name = 'Biology');*

Set Comparison

- The sub query *SELECT salary* FROM *instructor* WHERE *dept_name = 'Biology');* generates the set of all salary values of all instructors in the Biology department.
- The > ***some comparison in the where clause of the outer select is true if the salary*** value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

Set Comparison

- Let us find the names of all instructors that have a salary value greater than that of each instructor in the Biology department. The construct > ***all corresponds to the phrase "greater than all." Using this*** construct, we write the query as follows:

  select *name*

  from *instructor*

  where *salary > all (select salary*

  from *instructor*

  where *dept_name = 'Biology');*

Set Comparison

- As another example of set comparisons, consider the query **"Find the departments that have the highest average salary."**
- We begin by writing a query to find all average salaries, and then nest it as a subquery of a larger query that finds those departments for which the average salary is greater than or equal to all average salaries:

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
from instructor
group by dept_name);
```

Sub Queries in From Clause

- SQL allows a subquery expression to be used in the **from clause. The key concept** applied here is that any **select-from-where expression returns a relation as a result** and, therefore, can be inserted into another **select-from-where anywhere that a** relation can appear.

Sub Queries in From Clause

- Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000."
- The traditional query is ;

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

- The sub query in from clause is;

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

INNER JOIN

- Equijoins may be specified by equating attribute names from two or more relations:

  Example:

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

  Or Equivalently,

```
SELECT    FNAME, ADDRESS
FROM      EMPLOYEE  INNER JOIN DEPARTMENT
          ON DNO = DNUMBER
WHERE     DEPARTMENT.DNAME='Research';
```

- The DNO attribute from EMPLOYEE is equated to DNUMBER attribute of DEPARTMENT for performing the join. The keyword INNER is optional

INNER JOIN

- Natural joins may be specified using the NATURAL JOIN construct. It automatically finds attributes having the same names for performing the join.

- Relations may be renamed to accommodate natural join using the AS construct.

- Example:

```
SELECT  FNAME, LNAME, ADDRESS
FROM    (EMPLOYEE NATURAL JOIN DEPARTMENT
         AS DEPT(DNAME, DNO, MSSN, MSDATE)
WHERE   DNAME='Research';
```

INNER JOIN

SELECT *name, course id*

FROM *instructor, teaches*

WHERE *instructor.ID= teaches.ID;*

- This query can be written more concisely using the natural-join operation in SQL as:

SELECT *name, course id*

FROM *instructor natural join teaches;*

OUTER JOIN

- This query can be written more concisely using the natural-join operation in SQL as:

```
SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name
FROM (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
ON E.Super_ssn = S.Ssn);

SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name
FROM (EMPLOYEE AS E RIGHT OUTER JOIN EMPLOYEE AS S
ON E.Super_ssn = S.Ssn);

SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name
FROM (EMPLOYEE AS E FULL OUTER JOIN EMPLOYEE AS S
ON E.Super_ssn = S.Ssn);
```

OUTER JOIN

- If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN).
- Consider;

Student(sid, fname, lname, cid)

Course(cid, cname,credits)

```
SELECT *
FROM Student NATURAL LEFT OUTER JOIN Course;

SELECT *
FROM Student NATURAL RIGHT OUTER JOIN Course;

SELECT *
FROM Student NATURAL FULL OUTER JOIN Course;
```

MULTIWAY JOIN

- It is also possible to *nest join specifications; that is, one of the tables in a join may* itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join.**

MULTIWAY JOIN

- Consider, 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.
- The simple query is;

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum = Dnumber AND Mgr_ssn = Ssn AND
Plocation = ‘Stafford’
```

- The multiway join is;

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
JOIN EMPLOYEE ON Mgr_ssn = Ssn)
WHERE Plocation = ‘Stafford’;
```

Various OUTER JOIN Notations

- Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators + =, = +, and + = + for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in Oracle.

CROSS JOIN

- Cross Join is Cartesian Product of two relations. A cross join or cartesian product is formed when every row from one table is joined to all rows in another.

```
SELECT    FNAME, ADDRESS
FROM      EMPLOYEE CROSS JOIN DEPARTMENT
```

- When CROSS JOIN is executed with WHERE clause then it is similar to the INNER JOIN with ON clause.

Aggregate Functions

- These functions operate on a collection (a set or multiset) of values of a column of a relation as input and return a single value. They include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG.**
- Example: Find the maximum salary and the average salary among all employees.

```
SELECT  MAX(Salary), AVG(Salary)
FROM    Employee
```

```
SELECT SUM (Salary) AS Total_Sal, MAX (Salary) AS Highest_Sal,
MIN (Salary) AS Lowest_Sal, AVG (Salary) AS Average_Sal
FROM EMPLOYEE;
```

- Retrieve the total number of employees in the 'Research' department.

```
SELECT  COUNT (*)
FROM    Employee, Department
WHERE   Dno = Dnumber AND Dname='Research'
```

Aggregate Functions

```
SELECT COUNT (*)
FROM EMPLOYEE;
```

- However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(*) because tuples instead of values are counted.

- Count the number of distinct salary values in the database.

```
SELECT COUNT (DISTINCT Salary)
FROM EMPLOYEE
```

Group By Clause

- In many cases, we want to apply the aggregate functions *to* subgroups of tuples in a relation rather than all tuples. Each subgroup of tuples consists of the set of tuples that have *the same value* for the *grouping attribute(s).* The function is applied to each subgroup independently.
- SQL has a **GROUP BY**-clause for specifying the grouping attributes, which must also appear in the *SELECT-clause*.
- Example: For each department, retrieve the department number and the number of employees in the department.

```
SELECT  Dno, COUNT (*)
FROM   Employee
GROUP BY  Dno
```

- *In above query, the Employee tuples are divided into groups-each group having the same value for the **grouping attribute Dno**. The COUNT function is applied to each such group of tuples separately.*

Group By Clause

- If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute.* For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result

Having Clause

- Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions.*
- The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples) .
- Example: **Return all departments having more than twenty employees, and show the number of employees and their average salary.**

```
SELECT      Dno, COUNT(*), AVG(Salary)
FROM        Employee
GROUP BY    Dno
HAVING      COUNT(*) > 20;
```

- *If a **where** clause and **having** clause appear in the same query, SQL applies the predicate in the **where** clause first. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause. SQL then applies **having** clause, if it is present, to each group; it removes the groups that do not satisfy the **having** clause predicate. The **select** clause uses the remaining groups to generate tuples of the result of the query.*

Summary of Select SQL Queries

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

SELECT <attribute list>
FROM <table list>
[WHERE <condition>]
[GROUP BY <grouping attribute(s)>]
[HAVING <group condition>]
[ORDER BY <attribute list>]

Summary of Select SQL Queries

- The SELECT-clause lists the attributes or functions to be retrieved
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- GROUP BY specifies grouping attributes
- HAVING specifies a condition for selection of groups
- ORDER BY specifies an order for displaying the result of a query
- A query is evaluated by first applying the FROM-clause followed by the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause. Conceptually, ORDER BY is applied at the end to sort the query results.
- The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause.

Modification of database

- **Insert:**
  - It is used to add one or more tuples to a relation
  - Attribute values should be listed in the same order as the attributes were specified in the CREATE TABLE command
  - **Eg:**

    INSERT INTO EMPLOYEE

    VALUES (102, “Peter”, “Crouch”, ‘9-5-1973’, “Rajajinagar Bangalore”,’M’,300000, 100, 5);

    It Inserts an entire EMPLOYEE record with corresponding values.

  - Specific attributes can be populated by explicitly specifying them by name:

    INSERT INTO EMPLOYEE(Fname,Address,Dno)

    VALUES (“Arun”, “Pokhara”, 5);

    The other attributes will get either NULL values or their DEFAULT values

Modification of database

- **Delete:**
  - Removes tuples from a relation.
  - Includes a WHERE-clause to select the tuples to be deleted.
  - The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE-clause.
  - Tuples are deleted from only one table at a time (unless CASCADE is specified on a referential integrity constraint).
  - A missing WHERE-clause specifies that all tuples in the relation are to be deleted; the table then becomes an empty table.

Modification of database

- **Delete:**
- **Examples:**
  - DELETE FROM EMPLOYEE
    WHERE LNAME='Brown’
  - DELETE FROM EMPLOYEE
    WHERE SSN='123456789’
  - DELETE FROM EMPLOYEE
    WHERE DNO IN (SELECT DNUMBER
    FROM DEPARTMENT
    WHERE DNAME='Research')
  - DELETE FROM EMPLOYEE

Modification of database

- **Update:**
  - Used to modify attribute values of one or more selected tuples.
  - A WHERE-clause selects the tuples to be modified.
  - An additional SET-clause specifies the attributes to be modified and their new values.
  - Each command modifies tuples *in the same relation.*
  - Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

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

Modification of database

- **Update:**
- Example: Give all employees in the 'Research' department a 10% raise in salary.

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

- In this request, the modified SALARY value depends on the original SALARY value in each tuple.
- The reference to the SALARY attribute on the right of = refers to the old SALARY value before modification.
- The reference to the SALARY attribute on the left of = refers to the new SALARY value after modification.

Views

- A **view in SQL terminology is a single table that is derived from other tables.** These other tables can be *base tables or previously defined views. A view does not necessarily* exist in physical form; it is considered to be a **virtual table, in contrast to base tables, whose tuples are always physically stored in the database. This limits the** possible update operations that can be applied to views, but it does not provide any limitations on querying a view.
- We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- To create a view we use the command:

  create view *v* **as** <query expression>;

  <query expression> is any legal expression

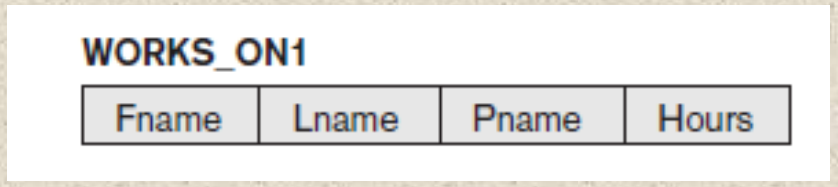
  v is the view name

Views

- For Example:

  Create View WORKS_ON1 as

  (Select Fname, Lname, Pname, Hours
  From EMPLOYEE, PROJECT, WORKS_ON
  Where SSN=ESSN AND Pnumber=Pno
  Group by Pname);



- **We can specify SQL queries on a newly create view:**

  SELECT Fname, lname from WORKS_ON1
  WHERE Pname='ProductX';

- **When no longer needed, a view can be dropped:**

  DROP WORKS_ON1;

Views

- For Example:

```
CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
AS SELECT Dname, COUNT (*), SUM (Salary)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber = Dno
GROUP BY Dname;
```

DEPT_INFO

Dept_name	No_of_emps	Total_sal

- The above VIEW explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

View Implementation

- The problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables.
- For the view;

```
SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname = 'ProductX';
```

- The above query would be automatically modified to the following query by the DBMS;

```
SELECT Fname, Lname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn = Essn AND Pno = Pnumber
AND Pname = 'ProductX';
```

View Implementation

- The disadvantage of the **query modification** is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time. The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied *to one of the defining base tables*. The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

View Implementation

- Different strategies as to when a materialized view is updated are possible. The **immediate update** strategy updates a view as soon as the base tables are changed; the **lazy update** strategy updates the view when needed by a view query; and the **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

Updating Views

- A view is supposed to be *always up-to-date*; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view does not have to be realized or materialized at the time of *view definition* but rather at the time when we *specify a query* on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept upto-date.

Updating Views

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have default values specified.*
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

Updating Views

- For Example:

```
Create View Emp_name as
(Select Fname, Lname
From EMPLOYEE);
```

```
UPDATE Emp_name
SET Fname= ‘Hari’
WHERE Lname = ‘Shrestha’;
```

Dropping Views

- If we do not need a view anymore, we can use the **DROP VIEW** command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement as;

  DROP VIEW WORKS_ON1;