MODULE 3

Chapter 1: SQL

4.1 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++.

4.2 SQL Data Definition and Data Types

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

4.2.1 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 .

For 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.

4.2.2 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) );
```

4.2.3 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
   - 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
   - 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.

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; for 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)
- For 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

4.3 Specifying Constraints in SQL

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

4.3.1 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 '888665555',
     -----------------
     -------------------
)
```

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.

4.3.2 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” relations 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.

4.3.3 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.

4.3.4 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 a manager’s start date is later than the department creation date

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

4.4 Basic Retrieval Queries in SQL

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

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

The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

SELECT <attribute list>

FROM <table list>

WHERE <condition>;

Where,

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

Examples:

1. Retrieve the birth date and 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='Research' **AND** Dnumber=Dno;

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

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

4.4.2 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 the 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, Sssn, Dno)
```

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

4.4.3 Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result.If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected.

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;
```

4.4.4 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.

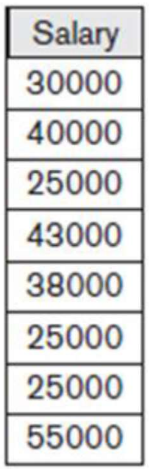
If we do want to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result.

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

(a) SELECT ALL Salary **FROM** EMPLOYEE;

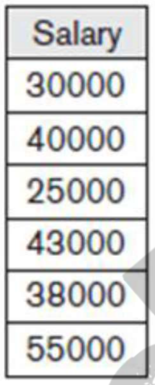
(b) SELECT DISTINCT Salary **FROM** EMPLOYEE;

(a)



Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)



Salary
30000
40000
25000
43000
38000
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 Lname='Smith' )
 UNION
( SELECT DISTINCT Pnumber FROM PROJECT, WORKS_ON,  EMPLOYEE
WHERE Pnumber=Pno AND Essn=Ssn AND   Lname='Smith' );
```

4.4.5 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 '%Houston,TX%';

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

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)).

4.4.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more 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 bylast 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
```

4.5 INSERT, DELETE, and UPDATE Statements in SQL

4.5.1The 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;

4.5.2 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.

4.5.3 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.

4.6 Additional Features of SQL

- SQL has various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.
- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases.
- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.
- SQL has language constructs for specifying the *granting and revoking of privileges* to users.
- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational**.
- SQL and relational databases can interact with new technologies such as XML

21. Briefly discuss how the different updata operations on a relation deal with constraint violations?

22. Consider the following schema for a COMPANY database:

    EMPLOYEE (Fname, Lname, Ssn, Address, Super-ssn, Salary, Dno)

    DEPARTMENT (Dname, Dnumber, Mgr-ssn, Mgr-start-date)

    DEPT-LOCATIONS (Dnumber, Dlocation)

    PROJECT (Pname, Pnumber, Plocation, Dnum)

    WORKS-ON (Ess!!, Pno, Hours)

    DEPENDENT (Essn, Dependent-name, Sex, Bdate, Relationship)

Write the queries in relational algebra.

i) Retrieve the name and address of all emp loyees who work for' Sales' department.

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

iii) List the names of all employees with two or more dependents.

iv) Retrieve the names of employees who have no dependents.

Chapter 2: SQL- Advances Queries

1.1 More Complex SQL Retrieval Queries

Additional features allow users to specify more complex retrievals from database

1.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value

Example

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute CollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

Each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used

Table 5.1 Logical Connectives in Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

The rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.

SQL allows queries that check whether an attribute value is NULL using the comparison operators **IS** or **IS NOT.**

Example: Retrieve the names of all employees who do not have supervisors.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL;
```

1.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**

Example1: List the project numbers of projects that have an employee with last name 'Smith' as manager

```
SELECT DISTINCT Pnumber  FROM PROJECT WHERE
 Pnumber IN
(SELECT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE
 WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='smith');
```

Example2: List the project numbers of projects that have an employee with last name 'Smith' as either manager or as worker.

```
SELECT DISTINCT Pnumber  FROM PROJECT WHERE
 Pnumber IN
(SELECT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='smith')
OR
Pnumber IN
(SELECT Pno FROM WORKS_ON, EMPLOYEE WHERE Essn=Ssn AND
 Lname='smith');
```

We make use of comparison operator **IN**, which compares a value *v* with a set (or multiset) of values *V* and evaluates to **TRUE** if *v* is one of the elements in *V*.

The first nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as manager. The second nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. For example, the following query will select the Essns of all employees who work the same (project, hours) combination on some project that employee ‘John Smith’ (whose Ssn = ‘123456789’) works on

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       (Pno, Hours) IN ( SELECT      Pno, Hours
                              FROM        WORKS_ON
                              WHERE       Essn=‘123456789’ );
```

In this example, the IN operator compares the subtuple of values in parentheses (Pno,Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

Nested Queries::Comparison Operators

Other comparison operators can be used to compare a single value v to a set or multiset *V.* The = ANY (or = SOME) operator returns TRUE if the value *v* is equal to *some value* in the set *V* and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (*v* > ALL *V*) returns TRUE if the value *v* is greater than *all* the values in the set (or multiset) *V*. For example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL ( SELECT Salary
FROM EMPLOYEE
WHERE Dno=5 );
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query,* and another in a relation in the FROM clause of the *nested query.* The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query**.

To avoid potential errors and ambiguities, create tuple variables (aliases) for all tables referenced in SQL query

Example: Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT Essn
FROM DEPENDENT AS D
WHERE E.Fname=D.Dependent_name
AND E.Sex=D.Sex );
```

In the above nested query, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex.

1.1.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**.

Example:

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT Essn
FROM DEPENDENT AS D
WHERE E.Fname=D.Dependent_name
AND E.Sex=D.Sex );
```

The nested query is evaluated once for each tuple (or combination of tuples) in the outer query. we can think of query in above example as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

1.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS Functions

The EXISTS function 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** if the nested query result contains at least one tuple, or
- **FALSE** if the nested query result contains no tuples.

For example, the query to retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee can be written using EXISTS functions as follows:

```
SELECT E.Fname, E.Lname
```

```
FROM EMPLOYEE AS E
WHERE EXISTS ( SELECT *
FROM DEPENDENT AS D
WHERE E.Ssn=D.Essn AND E.Sex=D.Sex
AND E.Fname=D.Dependent_name);
```

Example: List the names of managers who have at least one dependent

```
 SELECT Fname, Lname
FROM EMPLOYEE
WHERE EXISTS ( SELECT *
FROM DEPENDENT
WHERE Ssn=Essn )
AND
EXISTS ( SELECT *
FROM DEPARTMENT
WHERE Ssn=Mgr_ssn );
```

In general, EXISTS(Q) returns **TRUE** if there is at least one tuple in the result of the nested query Q, and it returns **FALSE** otherwise.

NOT EXISTS Functions

NOT EXISTS(Q) 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 EMPLOYEE
WHERE NOT EXISTS ( SELECT *
FROM DEPENDENT
WHERE Ssn=Essn );
```

For each EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

Example: Retrieve the name of each employee who works on all the projects controlled by department number 5

```
SELECT Fname, Lname
```

```
FROM EMPLOYEE
WHERE NOT EXISTS ( ( SELECT Pnumber
FROM PROJECT
WHERE Dnum=5)
EXCEPT ( SELECT Pno
FROM WORKS_ON
WHERE Ssn=Essn) );
```

UNIQUE Functions

UNIQUE(Q) returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

1.1.5 Explicit Sets and Renaming of Attributes in SQL

IN SQL it is possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses.

Example: Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE Pno IN (1, 2, 3);
```

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name

Example: Retrieve the last name of each employee and his or her supervisor

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

1.1.6 Joined Tables in SQL and Outer Joins

An SQL join clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two tables by using values common to each. SQL specifies four types of JOIN

1. INNER,
2. OUTER
3. EQUIJOIN and
4. NATURAL JOIN

INNER JOIN

An inner join is the most common join operation used in applications and can be regarded as the default join-type. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join- predicate (the condition). The result of the join can be defined as the outcome of first taking the Cartesian product (or Cross join) of all records in the tables (combining every record in table A with every record in table B)—then return all records which satisfy the join predicate

Example: **SELECT** * **FROM** employee

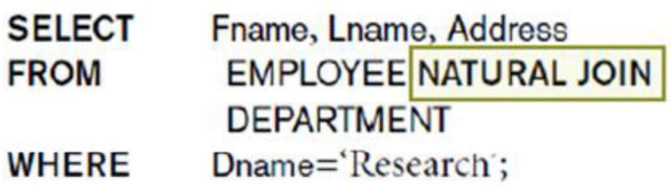
INNER JOIN department **ON**

employee.dno = department.dnumber;

EQUIJOIN and NATURAL JOIN

An **EQUIJOIN** is a specific type of comparator-based join that uses only equality comparisons in the join-predicate. Using other comparison operators (such as <) disqualifies a join as an equijoin.

NATURAL JOIN is a type of EQUIJOIN where the join predicate arises implicitly by comparing all columns in both tables that have the same column-names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns.



```
SELECT    Fname, Lname, Address
FROM       EMPLOYEE NATURAL JOIN
           DEPARTMENT
WHERE     Dname='Research';
```

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause.

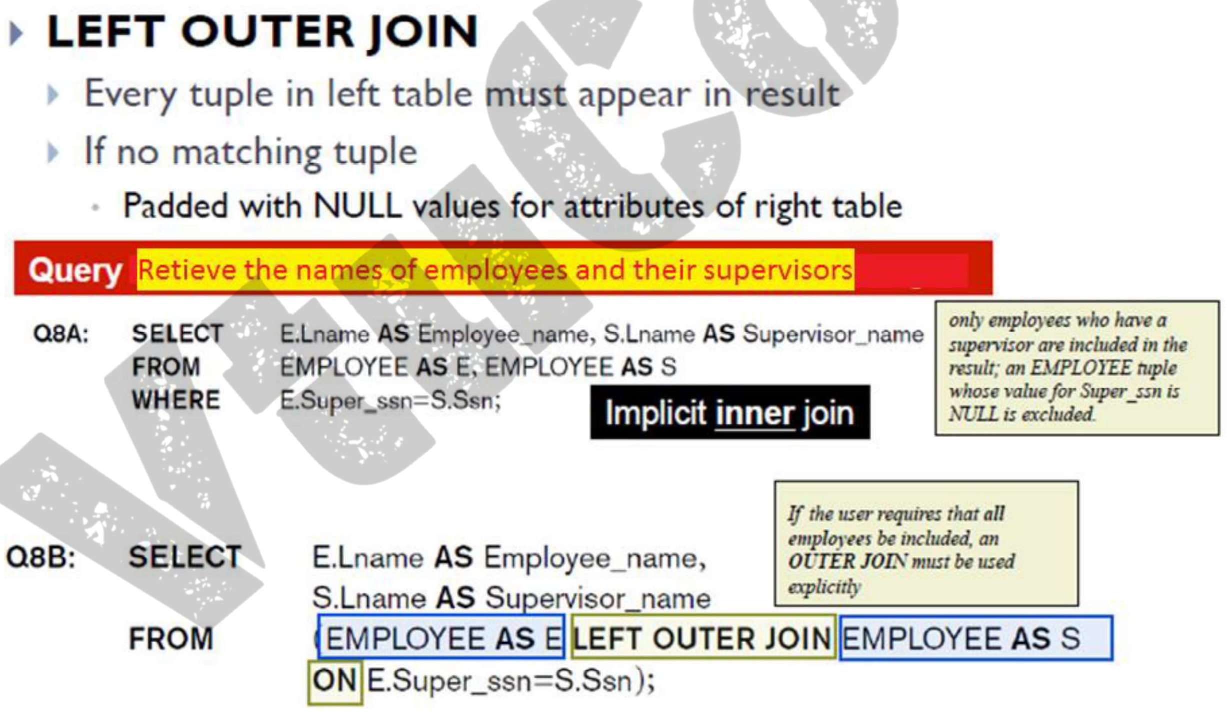
CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.

OUTER JOIN

An outer join does not require each record in the two joined tables to have a matching record. The joined table retains each record-even if no other matching record exists**.** Outer joins subdivide further into

- Left outer joins
- Right outer joins
- Full outer joins

No implicit join-notation for outer joins exists in standard SQL.



LEFT OUTER JOIN

- Every tuple in left table must appear in result
- If no matching tuple
  - Padded with NULL values for attributes of right table

Query Retieve the names of employees and their supervisors

```
Q8A:  SELECT   E.Lname AS Employee_name, S.Lname AS Supervisor_name
      FROM     EMPLOYEE AS E, EMPLOYEE AS S
      WHERE    E.Super_ssn=S.Ssn;
```

Implicit inner join

only employees who have a supervisor are included in the result; an EMPLOYEE tuple whose value for Super_ssn is NULL is excluded.

If the user requires that all employees be included, an OUTER JOIN must be used explicitly

```
Q8B:  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);
```

- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for the attributes of left table
- FULL OUTER JOIN
  - a full outer join combines the effect of applying both left and right outer joins.
  - Where records in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row.
  - For those records that do match, a single row will be produced in the result set (containing fields populated from both tables).

- 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
- For example, this syntax is available in Oracle. To specify the left outer join in Q8B using this syntax, we could write the query Q8C as follows:

```
Q8C:    SELECT    E.Lname, S.Lname
        FROM      EMPLOYEE E, EMPLOYEE S
        WHERE     E.Super_ssn += S.Ssn;
```

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**.

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.

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

1.1.7 Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG.** The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECTclause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another.

Examples

1. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM EMPLOYEE;
```

2. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
 FROM (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
 WHERE Dname='Research';
```

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

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

4. To retrieve the names of all employees who have two or more dependents

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE ( SELECT COUNT (*)
FROM DEPENDENT
WHERE Ssn=Essn ) >= 2;
```

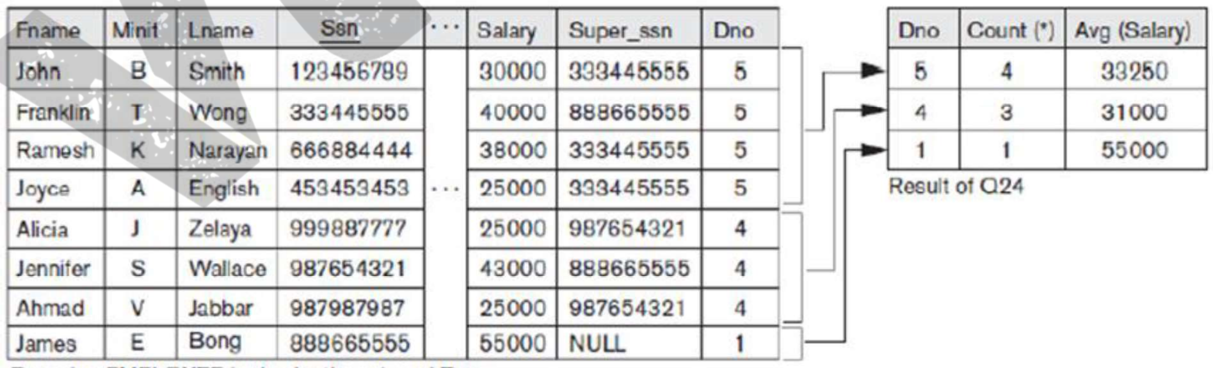
1.1.8 Grouping: The GROUP BY and HAVING Clauses

Grouping is used to create subgroups of tuples before summarization. For example, we may want to find the average salary of employees *in each department* or the number of employees who work *on each project*. In these cases we need to **partition** the relation into non overlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**.

SQL has a **GROUP BY** clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should *also appear in the SELECT clause,* so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

Example: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT Dno, COUNT (*), AVG (Salary)
FROM EMPLOYEE
GROUP BY Dno;
```



Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

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 of query

Example: For each project, retrieve the project number, the project name, and the number of employees who work on that project.

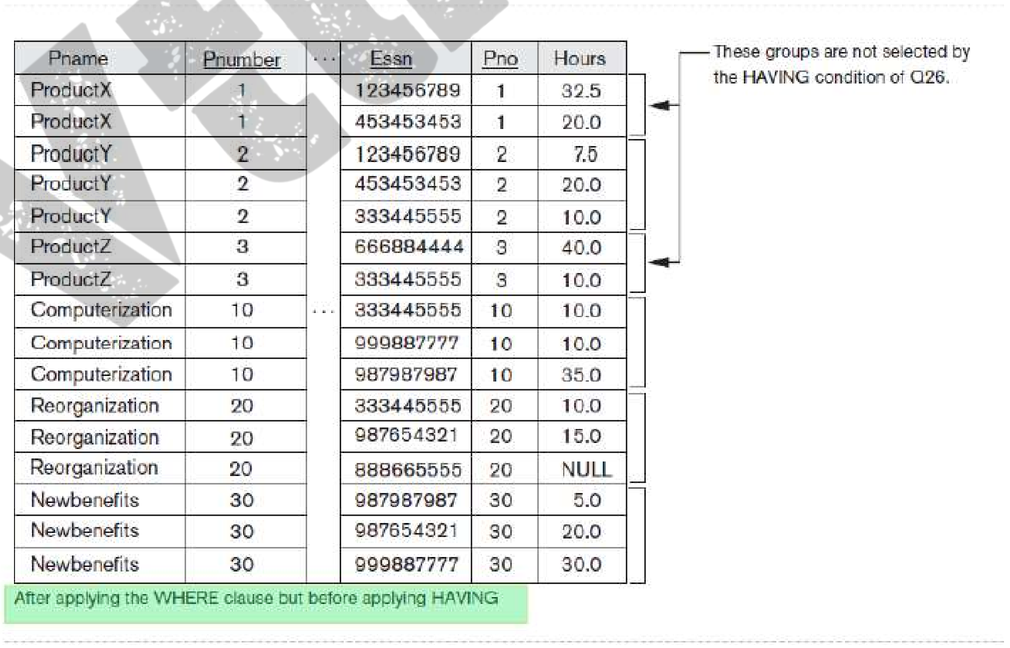
SELECT Pnumber, Pname, **COUNT** (*)
FROM PROJECT, WORKS_ON
WHERE Pnumber=Pno
GROUP BY Pnumber, Pname;

Above query shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations.

HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

Example: For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

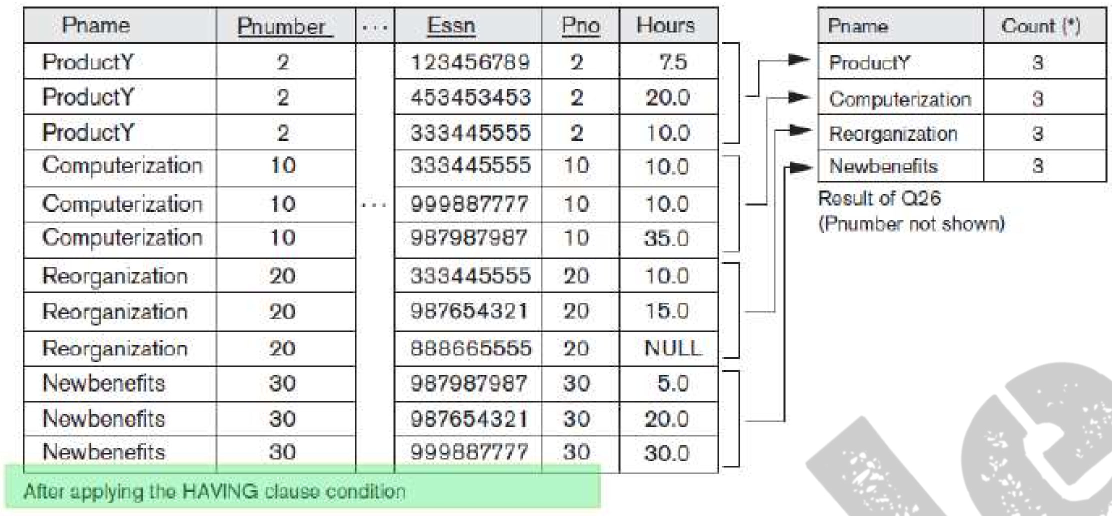
SELECT Pnumber, Pname, **COUNT** (*)
FROM PROJECT, WORKS_ON
WHERE Pnumber=Pno
GROUP BY Pnumber, Pname
HAVING COUNT (*) > 2;



Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING



Pname	Pnumber	. . .	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	. . .	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

Example: For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
SELECT Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno AND Ssn=Essn AND Dno=5
GROUP BY Pnumber, Pname;
```

Example: For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
SELECT Dnumber, COUNT (*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno AND Salary>40000 AND
( SELECT Dno
FROM EMPLOYEE
GROUP BY Dno
HAVING COUNT (*) > 5);
```

1.1.9 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory.The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way.The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

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

The **SELECT** clause lists the attributes or functions to be retrieved. The **FROM** clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The **WHERE** clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. **GROUP** BY specifies grouping attributes, whereas **HAVING** specifies a condition on the groups being selected rather than on the individual tuples. Finally, **ORDER BY** specifies an order for displaying the result of a query.

A query is evaluated conceptually by first applying the FROM clause to identify all tables involved in the query or to materialize any joined tables followed by the WHERE clause to select and join tuples, and then by GROUP BY and HAVING. ORDER BY is applied at the end to sort the query result Each DBMS has special query optimization routines to decide on an execution plan that is efficient to execute

In general, there are numerous ways to specify the same query in SQL.This flexibility in specifying queries has advantages and disadvantages.

- The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison. From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.
- The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way

1.2 Specifying Constraints as Assertions and Actions as Triggers

1.2.1 Specifying General Constraints as Assertions in SQL

Assertions are used to specify additional types of constraints outside scope of built-in relational model constraints. In SQL, users can specify general constraints via declarative assertions, using the **CREATE ASSERTION** statement of the DDL.Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

General form :

```
CREATE ASSERTION  <Name_of_assertion> CHECK (<cond>)
```

For the assertion to be satisfied, the condition specified after CHECK clause must return true.

For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for in SQL, we can write the following assertion:

```
 CREATE ASSERTION SALARY_CONSTRAINT
 CHECK ( NOT EXISTS ( SELECT * FROM EMPLOYEE E, EMPLOYEE M,
DEPARTMENT D  WHERE E.Salary>M.Salary  AND
 E.Dno=D.Dnumber  AND D.Mgr_ssn=M.Ssn ) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.

By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty

Example: consider the bank database with the following tables

- *branch (branch_name, branch_city, assets)*
- *customer (customer_name, customer_street, customer_city)*
- *account (account_number, branch_name, balance)*
- *loan (loan_number, branch_name, amount)*
- *depositor (customer_name, account_number)*
- *borrower (customer_name, loan_number)*

1. Write an assertion to specify the constraint that the Sum of loans taken by a customer does not exceed 100,000

```
CREATE ASSERTION sumofloans
CHECK (100000> = ALL
SELECT customer_name,sum(amount)
FROM borrower b, loan l
WHERE b.loan_number=l.loan_number
GROUP BY customer_name );
```

2. Write an assertion to specify the constraint that the Number of accounts for each customer in a given branch is at most two

```
CREATE ASSERTION NumAccounts
CHECK (   2 >= ALL
SELECT  customer_name,branch_name, count(*)
FROM     account A , depositor D
WHERE  A.account_number = D.account_number
GROUP BY customer_name, branch_name );
```

1.2.2 Introduction to Triggers in SQL

A trigger is a procedure that runs automatically when a certain event occurs in the DBMS. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. The CREATE TRIGGER statement is used to implement such actions in SQL.

General form:

```
CREATE TRIGGER <name>
BEFORE | AFTER | <events>
FOR EACH ROW |FOR EACH STATEMENT
WHEN (<condition>)
<action>
```

A trigger has three components

1. **Event:** When this event happens, the trigger is activated
   - Three event types : Insert, Update, Delete
   - Two triggering times: Before the event
     After the event

2. **Condition (optional):** If the condition is true, the trigger executes, otherwise skipped
3. **Action:** The actions performed by the trigger

When the **Event** occurs and **Condition** is true, execute the **Action**

Create Trigger *ABC*
Before Insert On
Students

This trigger is activated when an insert statement is issued, but before the new record is inserted

Create Trigger *XYZ*
After Update On Students
….

This trigger is activated when an update statement is issued and after the update is executed

Does the trigger execute for each updated or deleted record, or once for the entire statement ?. We define such granularity as follows:

Create Trigger *<name>*
Before| After   Insert| Update| Delete

For Each Row | For Each Statement
….

This is the event

This is the granularity

Create Trigger *XYZ*
After Update ON <tablename>
For each statement
….

This trigger is activated once (per UPDATE statement) after all records are updated

Create Trigger XYZ
Before Delete ON <tablename>
For each row
….

This trigger is activated before deleting each record

In the action, you may want to reference:

- The new values of inserted or updated records ***(:new)***
- The old values of deleted or updated records ***(:old)***

```
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary >150,000)
Begin
        if (:new.salary < 100,000) …
End;
```

Inside "When", the "new" and "old" should not have ":"

Trigger body

Inside the trigger body, they should have ":"

Examples:

1) If the employee salary increased by more than 10%, then increment the rank field by 1.

In the case of **Update** event only, we can specify which columns

```
Create Trigger EmpSal
Before Update Of salary On Employee
For Each Row
Begin
        IF (:new.salary > (:old.salary * 1.1)) Then
           :new.rank := :old.rank + 1;
        End IF;
End;
/
```

We changed the new value of ***rank*** field

The assignment operator has ":"

2) Keep the bonus attribute in Employee table always 3% of the salary attribute

```
Create Trigger EmpBonus
Before Insert Or Update On Employee
For Each Row
Begin
     :new.bonus := :new.salary * 0.03;
End;
```

Indicate two events at the same time

The bonus value is always computed automatically

3. Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database
   - Several events can trigger this rule:
     - inserting a new employee record
     - changing an employee's salary or
     - changing an employee's supervisor

   - Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION which will notify the supervisor

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN ) )
INFORM_SUPERVISOR(NEW.Supervisor_ssn,NEW.Ssn );
```

   - The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later
   - In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor
   - The action is to execute the stored procedure INFORM_SUPERVISOR

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates.

Assertions vs. Triggers

- Assertions do not modify the data, they only check certain conditions. Triggers are more powerful because the can check conditions and also modify the data
- Assertions are not linked to specific tables in the database and not linked to specific events. Triggers are linked to specific tables and specific events
- All assertions can be implemented as triggers (one or more). Not all triggers can be implemented as assertions

Example: Trigger vs. Assertion

All new customers opening an account must have opening balance >= $100. However, once the account is opened their balance can fall below that amount.

We need triggers, assertions cannot be used

Trigger Event: Before Insert

```
Create Trigger OpeningBal
Before Insert On Customer
For Each Row
Begin
    IF (:new.balance is null or :new.balance < 100) Then
     RAISE_APPLICATION_ERROR(-20004, 'Balance should be >= $100');
    End IF;
End;
```

1.3 Views (Virtual Tables) in SQL

1.3.1 Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables. 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.

For example, referring to the COMPANY database, we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE,WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE,WORKS_ON, and PROJECT tables the **defining tables** of the view.

1.3.2 Specification of Views in SQL

In SQL, the command to specify a view is **CREATE VIEW.** The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

Example 1:

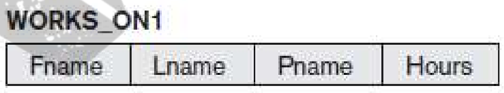
```
CREATE VIEW WORKS_ON1
 AS SELECT Fname, Lname, Pname, Hours
 FROM EMPLOYEE, PROJECT, WORKS_ON
 WHERE Ssn=Essn AND Pno=Pnumber;
```

Example 2:

```
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;
```

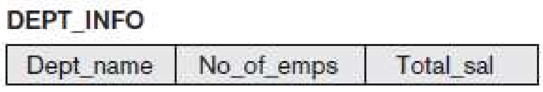
In example 1, we did not specify any new attribute names for the view WORKS_ON1. In this case,WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

Example 2 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.



WORKS_ON1

Fname	Lname	Pname	Hours



DEPT_INFO

Dept_name	No_of_emps	Total_sal

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables.

For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as :

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

The same query would require the specification of two joins if specified on the base relations directly. one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism.

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 is not 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 up-to-date.

If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it. For example : **DROP VIEW** WORKS_ON1;

1.3.3 View Implementation, View Update and Inline Views

The problem of efficiently implementing a view for 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 example, the query

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

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';
```

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple 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 view table when the view is first queried 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.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways. Hence, it is often not possible for the DBMS to determine which of the updates is intended.

To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1: UPDATEWORKS_ON1
     SET Pname = 'ProductY'
     WHERE Lname='Smith' AND Fname='John'
     AND Pname='ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create additional side effects that affect the result of other queries.

For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(a): UPDATEWORKS_ON
   SET Pno= (SELECT Pnumber
    FROM PROJECT
    WHERE Pname='ProductY' )
    WHERE Essn IN ( SELECT Ssn
    FROM EMPLOYEE
   WHERE Lname='Smith' AND Fname='John' )
    AND
    Pno= (SELECT Pnumber
    FROM PROJECT
   WHERE Pname='ProductX' );
```

```
(b): UPDATEPROJECT SET Pname = 'ProductY'
     WHERE Pname = 'ProductX';
```

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'.

It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view does not make sense because Total_sal is defined to be the sum of the individual employee salaries. This request is shown as UV2:

```
UV2: UPDATEDEPT_INFO
     SET Total_sal=100000
     WHERE Dname='Research';
```

A large number of updates on the underlying base relations can satisfy this view update.

Generally, a view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to more than one update on the underlying base relations, we must have a certain procedure for choosing one of the possible updates as the most likely one.

In summary, we can make the following observations:

- 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.

In SQL, the clause **WITH CHECK OPTION** must be added at the end of the view definition if a view *is to be updated*. This allows the system to check for view updatability and to plan an execution strategy for view updates. It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

1.4 Schema Change Statements in SQL

Schema evolution commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema.

1.4.1 The DROP Command

The DROP command can be used to drop named schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is no longer needed, the DROP SCHEMA command can be used.

There are two drop behavior options: **CASCADE** and **RESTRICT**. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY **CASCADE**;

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

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 get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT **CASCADE**;

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

The DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command should be used instead of DROP TABLE.

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

1.4.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible **alter table actions** include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.

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

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

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple. If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is not allowed in this case.

To drop a column, we must choose either **CASCADE** or **RESTRICT** for drop behavior. If **CASCADE** is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If **RESTRICT** is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column.

For example, the following command removes the attribute Address from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;

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

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

ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **SET DEFAULT** '333445555';

Alter Table - Alter/Modify Column

To change the data type of a column in a table, use the following syntax:

ALTER TABLE table_name
MODIFY column_name datatype;

For example we can change the data type of the column named "DateOfBirth" from date to year in the "Persons" table using the following SQL statement:

ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

Chapter 3: Database Application Development

2.1 Introduction

We often encounter a situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL.For example, we may want to integrate a database applications with GUI or we may want to integrate with other existing applications.

2.2 Accessing Databases from applications

SQL commands can be executed from within a program in a host language such as C or Java. A language to which SQL queries are embedded are called Host language.

2.2.1 Embedded SQL

The use of SQL commands within a host language is called **Embedded SQL**. Conceptually, embedding SQL commands in a host language program is straight forward. SQL statements can be used wherever a statement in the host language is allowed. SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Any host language variable used to pass arguments into an SQL command must be declared in SQL.

There are two complications:

1. Data types recognized by SQL may not be recognized by the host language and vice versa
   - This mismatch is addressed by casting data values appropriately before passing them to or from SQL commands.
2. SQL is set-oriented
   - Addressed using cursors

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host language variables must be prefixed by a colon(:) in SQL statements and be declared between the commands

EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION

The declarations are similar to C, are separated by semicolons. For example, we can declare variables c_sname, c_sid, c_rating, and c_age (with the initial c used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION

        char c_sname[20];
        long c_sid;
        short c_rating;
        float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, c_sname has the type CHARACTER(20) when referred to in an SQL statement, c_sid has the type INTEGER, crating has the type SMALLINT, and c_age has the type REAL.

We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, **SQLCODE** and **SQLSTATE**.

- **SQLCODE** is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes.
- **SQLSTATE**, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported.

One of these two variables must be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char [6] , that is, a character string five characters long.

Embedding SQL statements

All SQL statements embedded within a host program must be clearly marked with the details dependent on the host language. In C, SQL statements must be prefixed by **EXEC SQL.** An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

Example: The following embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the sailors relation

```
EXEC SQL INSERT INTO sailors VALUES (:c_sname, :c_sid, :c_rating,:c_age);
```

The **SQLSTATE** variable should be checked for errors and exceptions after each Embedded SQL statement.SQL provides the **WHENEVER** command to simplify this task:

EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [CONTINUE|GOTO stmt]

If **SQLERROR** is specified and the value of SQLSTATE indicates an exception, control is transferred to stmt, which is presumably responsible for error and exception handling. Control is also transferred to stmt if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

2.2.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on sets of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation- this mechanism is called a **cursor**

We can declare a cursor on any relation or on any SQL query. Once a cursor is declared, we can

- **open** it (positions the cursor just before the first row)
- **Fetch** the next row
- **Move** the cursor (to the next row,to the row after the next n, to the first row or previous row etc by specifying additional parameters for the fetch command)
- **Close** the cursor

Cursor allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

- We usually need to open a cursor if the embedded statement is a SELECT. we can avoid opening a cursor if the answer contains a single row
- INSERT, DELETE and UPDATE statements require no cursor. some variants of DELETE and UPDATE use a cursor.

Examples:

i) Find the name and age of a sailor, specified by assigning a value to the host variable c_sid, declared earlier

```
EXEC SQL SELECT s.sname,s.age
INTO  :c_sname, :c_age
FROM Sailaor s
WHERE s.sid=:c.sid;
```

The **INTO** clause allows us assign the columns of the single answer row to the host variable c_sname and c_age. Therefore, we do not need a cursor to embed this query in a host language program.

ii) Compute the name and ages of all sailors with a rating greater than the current value of the host variable c_minrating

SELECT s.sname,s.age
FROM sailors s **WHERE** s.rating>:c_minrating;

The query returns a collection of rows. The INTO clause is inadequate. The solution is to use a cursor:

DECLARE sinfo **CURSOR FOR**
SELECT s.sname,s.age
FROM sailors s
WHERE s.rating>:c_minrating;

This code can be included in a C program and once it is executed, the cursor sinfo is defined.
We can open the cursor by using the syntax:

OPEN sinfo;

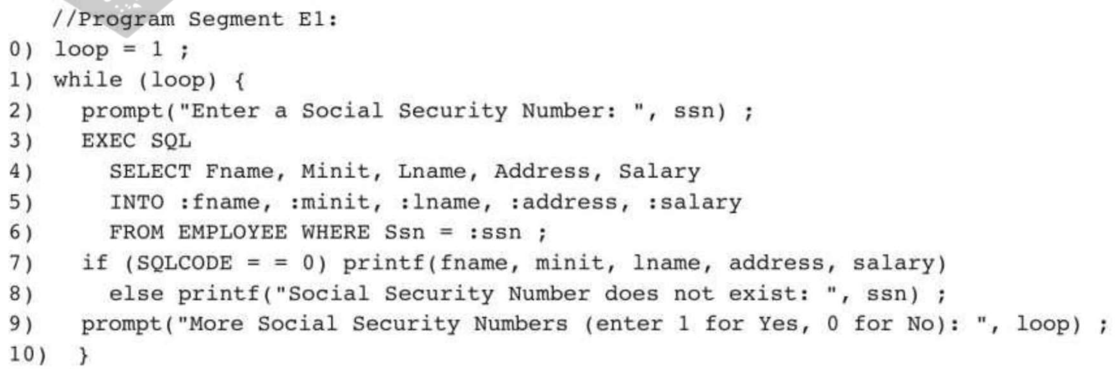
A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it.When the cursor is opened, it is positioned just before the first row.
We can use the FETCH command to read the first row of cursor sinfo into host language variables:

FETCH sinfo INTO :c_sname, :c_age;

When the FETCH statement is executed, the cursor is positioned to point at the next row and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement, we can read all the rows computed by the query, one row at time.
When we are done with a cursor, we can close it:

CLOSE sinfo;

iii) To retrieve the name, address and salary of an employee specified by the variable ssn



```
//Program Segment E1:
loop = 1 ;
while (loop) {
  prompt("Enter a Social Security Number: ", ssn) ;
  EXEC SQL
    SELECT Fname, Minit, Lname, Address, Salary
    INTO :fname, :minit, :lname, :address, :salary
    FROM EMPLOYEE WHERE Ssn = :ssn ;
  if (SQLCODE = = 0) printf(fname, minit, lname, address, salary)
    else printf("Social Security Number does not exist: ", ssn) ;
  prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
 }
```

Properties of Cursors

The general form of a cursor declaration is:

DECLARE *cursorname* **[INSENSITIVE] [SCROLL] CURSOR**
[WITH HOLD]
FOR *some query*
[ORDER BY order-item-list **]**
[FOR READ ONLY I FOR UPDATE]

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or updatable cursor (FOR UPDATE).If it is updatable, simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if sinfo is an updatable cursor and open, we can execute the following statement:

UPDATE Sailors S
SET S.rating = S.rating -1
WHERE CURRENT of sinfo;

A cursor is updatable by default unless it is a scrollable or insensitive cursor in which case it is read-only by default.

If the keyword **SCROLL** is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed

If the keyword **INSENSITIVE** is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior.

A holdable cursor is specified using the WITH **HOLD** clause, and is not closed when the transaction is committed.

Optional **ORDER BY** clause can be used to specify a sort order. The order-item-list is a list of order-items. An order-item is a column name, optionally followed by one of the keywords ASC or DESC Every column mentioned in the **ORDER BY** clause must also appear in the select-list of the query associated with the cursor; otherwise it is not clear what columns we should sort on

ORDER BY minage **ASC**, rating **DESC**

The answer is sorted first in ascending order by minage, and if several rows have the same minage value, these rows are sorted further in descending order by rating

Rating	minage
8	25.5
3	25.5
7	35.0

Dynamic SQL

Dynamic SQL Allow construction of SQL statements on-the-fly. Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed. SQL provides some facilities to deal with such situations; these are referred to as **Dynamic SQL**.

Example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :csqlstring;
EXEC SQL EXECUTE readytogo;
```

- The first statement declares the C variable *c_sqlstring* and initializes its value to the string representation of an SQL command
- The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable *readytogo*
- The third statement executes the command

2.3 An Introduction to JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. A DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. The details of this translation vary across DBMSs, and therefore even though the source code can be compiled to work with different DBMSs, the final executable works only with one specific DBMS.

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language.

- In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *Without recompilation.*

- While Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable
- In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously
- ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection
- All direct interaction with a specific DBMS happens through a DBMS-specific driver.

A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a driver manager a driver does not necessarily need to interact with a DBMS that understands SQL. It is sufficient that the driver translates the SQL commands from the application into equivalent commands that the DBMS understands.

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source. There is no limit on the number of open connections. An application can have several open connections to different data sources. Each connection has transaction semantics; that is, changes from one connection are visible to other connections only after the connection has committed its changes. While a connection is open, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.

2.3.1 Architecture

The architecture of JDBC has four main components:

- Application
- Driver manager
- Drivers
- Data sources

Application

- initiates and terminates the connection with a data source
- sets transaction boundaries, submits SQL statements and retrieves the results

Driver manager

- Load JDBC drivers and pass JDBC function calls from the application to the correct driver
- Handles JDBC initialization and information calls from the applications and can log all function calls
- Performs some rudimentary error checking

Drivers

- Establishes the connection with the data source
- Submits requests and returns request results
- Translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard

Data sources

- Processes commands from the driver and returns the results

Drivers in JDBC are classified into four types depending on the architectural relationship between the application and the data source:

Type I Bridges:

- This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS.
- An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge.
- Advantage:
  - it is easy to piggyback the application onto an existing installation, and no new drivers have to be installed.
- Drawbacks:
  - The increased number of layers between data source and application affects performance
  - the user is limited to the functionality that the ODBC driver supports.

Type II Direct Translation to the Native API via Non-Java Driver:

- This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source.
- The driver is usually ,written using a combination of C++ and Java; it is dynamically linked and specific to the data source.
- Advantage
    - This architecture performs significantly better than a JDBC-ODBC bridge.
- Disadvantage
    - The database driver that implements the API needs to be installed on each computer that runs the application.

Type III~~Network Bridges:

- The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations.
- In this case, the driver on the client site is not DBMS-specific.
- The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server.
- The middleware server can then use a Type II JDBC driver to connect to the data source.

Type IV-Direct Translation to the Native API via Java Driver:

- Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets
- In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system.
- This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

2.4 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from programs written in the Java language. It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling.

The classes and interfaces are part of the java.sql package. JDBC 2.0 also includes the javax.sql package, the JDBC Optional Package. The package javax.sql adds, among other things, the capability of connection pooling and the Row-Set interface.

2.4.1 JDBC Driver Management

In JDBC, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers. The Drivermanager class has methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. The following Java example code explicitly loads a JDBC driver:

Class.forName("oracle/jdbc.driver.OracleDriver");

There are two other ways ofregistering a driver. We can include the driver with -Djdbc. drivers=oracle/jdbc. driver at the command line when we start the Java application. Alternatively, we can explicitly instantiate a driver, but this method is used only rarely, as the name of the driver has to be specified in the application code, and thus the application becomes sensitive to changes at the driver level.

After registering the driver, we connect to the data source.

2.4.2 Connections

A session with a data source is started through creation of a Connection object; Connections are specified through a JDBC URL, a URL that uses the jdbc protocol. Such a URL has the form

jdbc:<subprotocol>:<otherParameters>

```
String uri = .. jdbc:oracle:www.bookstore.com:3083..
Connection connection;
try
 {
        Connection connection =
        DriverManager.getConnection(url, userId,password);
    }
catch(SQLException excpt)
{
        System.out.println(excpt.getMessageO);
        return;
}
```

Program code: Establishing a Connection with JDBC

In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If **autocommit** is set for a connection, then each SQL statement is considered to be its own transaction. If **autocommit** is off, then a series of statements that compose a transaction can be committed using the commit() method of the Connection class, or aborted using the rollback() method. The Connection class has methods to set the autocommit mode (Connection. setAutoCommit) and to retrieve the current autocommit mode (getAutoCommit). The following methods are part of the Connection interface and permit setting and getting other properties:

- public int getTransactionIsolation() throws SQLException and
  public void setTransactionIsolation(int 1) throws SQLException.
  - These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation are possible, and argument *l* can be set as follows:
    - TRANSACTION_NONE
    - TRANSACTION_READ_UNCOMMITTED
    - TRANSACTION_READ_COMMITTED
    - TRANSACTION_REPEATABLE_READ
    - TRANSACTION_SERIALIZABLE
- public boolean getReadOnlyO throws SQLException and
  public void setReadOnly(boolean readOnly) throws SQLException.
  - These two functions allow the user to specify whether the transactions executecl through this connection are rcad only.
- public boolean isClosed() throws SQLException.
  - Checks whether the current connection has already been closed.
- setAutoCommit and get AutoCommit.

In case an application establishes many different connections from different parties (such as a Web server), connections are often **pooled** to avoid this overhead. A **connection pool** is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source.

2.4.3 Executing SQL Statements

JDBC supports three different ways of executing statements:

- Statement
- PreparedStatement, and
- CallableStatement.

The **Statement** class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.

The **PreparedStatement** class dynamically generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object is created.

```
/ / initial quantity is always zero
String sql = "INSERT INTO Books VALUES('?, 7, '?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
/ / now instantiate the parameters with values
/ / a,ssume that isbn, title, etc. are Java variables that
/ / contain the values to be inserted
pstmt.clearParameters() ;
pstmt.setString(l, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);
int numRows = pstmt.executeUpdate();
```

program code: SQL Update Using a PreparedStatement Object

The SQL query specifies the query string, but uses ''?' for the values of the parameters, which are set later using methods setString, setFloat,and setInt. The ''?" placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements. The method setString is one way to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters() before setting parameter values in order to remove any old data.

There are different ways of submitting the query string to the data source. In the example, we used the **executeUpdate** command, which is used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT,ALTER, and DELETE statements). The executeUpdate method returns

- an integer indicating the number of rows the SQL statement modified;
- 0 for successful execution without modifying any rows.

The executeQuery method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object.

2.4.4 ResultSets

ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions. In its most basic form, the **ResultSet** object allows us to read one row of the output of the query at a time. Initially, the **ResultSet** is positioned before the first row, and we have to retrieve the first row with an explicit call to the **next()** method. The next method returns false if there are no more rows in the query answer, and true other\vise. The code fragment shown below illustrates the basic usage of a ResultSet object:

```
ResultSet rs=stmt.executeQuery(sqlQuery);
/ / rs is now a cursor
/ / first call to rs.nextO moves to the first record
/ / rs.nextO moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next())
 {
     / / process the data
 }
```

While next () allows us to retrieve the logically next row in the query answer, we can move about in the query answer in other ways too:

- **previous()** moves back one row.
- **absolute(int num)** moves to the row with the specified number.
- **relative(int num)** moves forward or backward (if num is negative) relative to the current position. **relative (-1)** has the same effect as previous.
- **first()** moves to the first row, and **last()** moves to the last row.

Matching Java and SQL Data Types

In considering the interaction of an application with a data source, the issues we encountered in the context of Embedded SQL (e.g., passing information between the application and the data source through shared variables) arise again. To deal with such issues, JDBC provides special data types and specifies their relationship to corresponding SQL data types. Table 2.4.4 shows the accessor methods in a ResultSet object for the most common SQL datatypes.

With these accessor methods, we can retrieve values from the current row of the query result referenced by the ResultSet object. There are two forms for each accessor method. One method retrieves values by column index, starting at one, and the other retrieves values by column name.

The following example shows how to access fields of the current ResultSet row using accesssor methods.

```
ResultSet rs=stmt.executeQuery(sqlQuery);

String sqlQuerYi
ResultSet rs = stmt.executeQuery(sqIQuery)
while (rs.nextO)
{
    isbn = rs.getString(l);
    title = rs.getString(" TITLE");
    / / process isbn and title
}
```

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Table 2.4.4 : Reading SQL Datatypes from a ResultSet Object

2.4.5 Exceptions and Warnings

Similar to the SQLSTATE variable, most of the methods in java. sql can throw an exception of the type SQLException if an error occurs. The information includes SQLState, a string that describes the error (e.g., whether the statement contained an SQL syntax error). In addition to the standard getMessage() method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

- public String getSQLState() returns an SQLState identifier based on the SQL:1999 specification
- public int getErrorCode () retrieves a vendor-specific error code.

- public SQLException getNextExceptionO gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQLWarning is a subclass of SQLException. Warnings are not as severe as errors and the program can usually proceed without special handling of warnings. Warnings are not thrown like other exceptions, and they are not caught as part of the try-catch block around a java.sql statement. We need to specifically test whether warnings exist. **Connection**, **Statement**, and **ResultSet** objects all have a **getWarnings()** method with which we can retrieve SQL warnings if they exist. Duplicate retrieval of warnings can be avoided through **clearWarnings(). Statement** objects clear warnings automatically on execution of the next statement; **ResultSet** objects clear warnings every time a new tuple is accessed.

Typical code for obtaining SQLWarnings looks similar to the code shown below:

```
try
{
        stmt = con.createStatement();
        warning = con.getWarnings();
        while( warning != null)
         {
                / / handleSQLWarnings / / code to process warning
                warning = warning.getNextWarningO; / /get next warning
        }
        con.clear\Varnings() ;
        stmt.executeUpdate( queryString );
        warning = stmt.getWarnings();
        while( warning != null)
         {
                / / handleSQLWarnings / / code to process warning
                warning = warning.getNextWarningO; / /get next warning
        }
} / / end try
catch ( SQLException SQLe)
 {
        / / code to handle exception
} / / end catch
```

2.4.6 Examining Database Metadata

We can use the DatabaseMetaData object to obtain information about the database system itself, as well as information from the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
Databa..seMetaData md = con.getMetaD<Lta():
System.out.println("Driver Information:");
System.out.println("Name:" + md.getDriverNameO
                    + "; version:" + mcl.getDriverVersion());
```

The DatabaseMetaData object has many more methods (in JDBC 2.0, exactly 134). Some of the methods are:

- **public ResultSet getCatalogs() throws SqLException**. This function returns a ResultSet that can be used to iterate over all **public int getMaxConnections() throws SqLException** the catalog relations.This function returns the maximum number of connections possible.

Example: code fragment that examines all database metadata

```
DatabaseMetaData dmd = con.getMetaDataO;
ResultSet tablesRS = dmd.getTables(null,null,null,null);
string tableName;
while(tablesRS.next())
 {
        tableNarne = tablesRS .getString("TABLE_NAME");
        / / print out the attributes of this table
        System.out.println("The attributes of table"
                    + tableName + " are:");
        ResultSet columnsRS = dmd.getColums(null,null,tableName, null);
        while (columnsRS.next())
        {
                System.out.print(colummsRS.getString("COLUMN_NAME")
                +" ");
        }
        / / print out the primary keys of this table
        System.out.println("The keys of table" + tableName + " are:");
        ResultSet keysRS = dmd.getPrimaryKeys(null,null,tableName);
        while (keysRS. next ())
        {
                System.out.print(keysRS.getStringC'COLUMN_NAME") +" ");
        }
```

}

7 steps for jdbc :

1. Import the package
   -- import java.sql.*;
2. Load and register the driver
   --class.forname();
3. Establish the connection
   -- Connection con;
4. Create a Statement object
   -- Statement st;
5. Execute a query
   -- st.execute();
6. Process the result
7. Close the connection

Step 2: load the corresponding JDBC driver

Class.forName("oracle/jdbc.driver.OracleDriver");

Step 3: create a session with data source through creation of Connection object.

```
Connection connection = DriverManager.getConnection(database_url,
                                    userId, password);
EX: Connection con= DriverManager.getConnection
                        ("jdbc:oracle:thin:@localhost:1521:xesid","system","ambika");
```

Step 4:create a statement object

- JDBC supports three different ways of executing statements:

- Statement
- PreparedStatement and
- CallableStatement.

- The Statement class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.
- The PreparedStatement class dynamically generates precompiled SQL statements that can be used several times
- CallableStatement are used to call stored procedures from JDBC. CallableStatement is a subclass of PreparedStatement and provides the same functionality.
- Example:

```
Statement st=con.createStatement();
```

Step 5: executing a query

String query=“select * from students where usn=‘4VV15CS001’”;

ResultSet rs=st.executeQuery(query);

Step 6: process the result

String sname=rs.getString(2);

System.out.println(sname);

Step 7: close the connection

con.close();

```
import java.sql.*;

public class Demo {

public static void main(String[] args) {

   try

     {

          String query="select * from students where usn=‘4VV15CS001’";

          Class.forName("oracle/jdbc.driver.OracleDriver");

          Connection con = DriverManager.getConnection

                              ("jdbc:oracle:thin:@localhost:1521:xesid","system","ambika");

          Statement st=con.createStatement();

          ResultSet rs=st.executeQuery(query);

         String s=rs.getString(1);

          System.out.println(s);

          con.close();

     }

     catch(Exception e)

     {

      }

  }
```

2.5 SQLJ: SQL-JAVA

SQLJ enables applications programmers to embed SQL statements in Java code in a way that is compatible with the Java design philosophy

Example: SQLJ code fragment that selects records from the Books table that match a given author.

```
String title; Float price; String author;
#sql iterator Books (String title, Float price);
Books books;
#sql books = {
        SELECT title, price INTO :title, :price
        FROM Books WHERE author = :author
      };
while (books.next()) {
        System.out.println(books.title() + ", " + books.price());
      }
books.close() ;
```

All SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class.

Usage of an iterator in SQLJ goes through five steps:

1. Declare the Iterator Class: In the preceding code, this happened through the statement

   #sql iterator Books (String title, Float price);

   This statement creates a new Java class that we can use to instantiate objects.
2. Instantiate an Iterator Object from the New Iterator Class:

   We instantiated our iterator in the statement Books books;.
3. Initialize the Iterator Using a SQL Statement:

   In our example, this happens through the statement #sql books =....
4. Iteratively, Read the Rows From the Iterator Object:

   This step is very similar to reading rows through a ResultSet object in JDBC.
5. Close the Iterator Object.

There are two types of iterator classes:

- named iterators
- positional iterators

For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name. This method is used in our example.

For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a FETCH ... INTO construct, similar to Embedded SQL

We can make the iterator a positional iterator through the following statement:

```
#sql iterator Books (String, Float);
```

We then retrieve the individual rows from the iterator as follows:

```
while (true)
  {
    #sql { FETCH :books INTO :title, :price, };
   if (books.endFetch())
   { break: }
  / / process the book
}
```

2.6 STORED PROCEDURES

Stored procedure is a set of logical group of SQL statements which are grouped to perform a specific task.

Benefits :

- reduces the amount of information transfer between client and database server
- Compilation step is required only once when the stored procedure is created. Then after it does not require recompilation before executing unless it is modified and reutilizes the same execution plan whereas the SQL statements need to be compiled every time whenever it is sent for execution even if we send the same SQL statement every time
- It helps in re usability of the SQL code because it can be used by multiple users and by multiple clients since we need to just call the stored procedure instead of writing the same SQL statement every time. It helps in reducing the development time

Syntax:

```
Create or replace procedure <procedure Name>  [(arg1 datatype, arg2 datatype)]
 Is/As
            <declaration>
 Begin
           <SQL Statement>
Exception
             -----
             -----
End procedurename;
```

2.6.1 Creating a Simple Stored Procedure

Consider the following schema:

Student(usn:string,sname:string)

Let us now write a stored procedure to retrieve the count of students with sname ‘Akshay’

```
create or replace procedure ss
  is
stu_cnt int;
begin
           select count(*) into stu_cnt from students where sname='AKSHAY';
           dbms_output.put_line('the count of student is :' || stu_cnt);
 end ss;
```

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT.

- IN parameters are arguments to the stored procedure
- OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process
- INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values

Example:

```
CREATE PROCEDURE AddInventory (
IN book_isbn CHAR(IO),
IN addedQty INTEGER)
UPDATE Books  SET qty_in_stock = qtyjn_stock + addedQty
WHERE bookjsbn = isbn
```

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
  char isbn[IO];
  long qty;
EXEC SQL END DECLARE SECTION
/ / set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Procedures without parameters are called **static procedures** and with parameters are called **dynamic procedures.**

Example: stored procedure with parameter

```
create or replace procedure emp(Essn int)
 as
      eName varchar(20);
 begin
       select fname into eName from employee where ssn=Essn and dno=5;
        dbms_output.put_line(' the employee name is   :'||Essn ||eName);
end emp;
```

2.6.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

```
CALL storedProcedureName(argl, arg2, .. ,argN);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class.A stored procedure could contain multiple SQL statements or a series of SQL statements-thus, the result could be many different ResultSet objects.We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt= con. prepareCall(" {call ShowNumberOfOrders}");
 ResultSet rs = cstmt.executeQuery();
  while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
/ / create the cursor class
        #sql Iterator CustomerInfo(int cid, String cname, int count);
/ / create the cursor
        CustomerInfo customerinfo;
        / / call the stored procedure
        #sql customerinfo = {CALL ShowNumberOfOrders};
        while (customerinfo.next()
         {
                System.out.println(customerinfo.cid() + "," +
                customerinfo.count()) ;
        }
```

2.6.3 SQL/PSM

SQL/Persistent Stored Modules is an ISO standard mainly defining an extension of SQL with procedural language for use in stored procedures.

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,... , parameterN)
local variable declarations
procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameterl, ... , parameterN)
RETURNS sqlDataType
local variable declarations
function code;
```

Example:

```
CREATE FUNCTION RateCustomer (IN custId INTEGER, IN year INTEGER)
RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders = (SELECT COUNT(*) FROM Orders 0 WHERE O.tid = custId);
IF (numOrders> 10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=O;
END IF;
RETURN rating;
```

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
ELSEIF statements;
ELSEIF statements;
ELSE statements;
END IF
```

- Loops are of the form

```
LOOP
        statements:
END LOOP
```

Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables.We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon ':'.