18CS53: DATABASE MANAGEMENT SYSTEMS
(Effective from the academic year 2018 -2019)

Semester: 05	**Credits: 04**
Number of Hours/Week: 03:02:0	**IA Marks: 40**
Total Number of Hours: 50	**Exam Marks: 60**

Module-III Notes

TOPICS

SQL: Advances Queries: More complex SQL retrieval queries
Specifying constraints as assertions and action triggers,
Views in SQL, Schema change statements in SQL.
Database Application Development: Accessing databases from applications
An introduction to JDBC, JDBC classes and interfaces,
SQLJ, Stored procedures,
Case study: The internet Bookshop
Internet Applications: The Three-Tier application architecture,
The presentation layer,
The Middle Tier

SQL: Advances Queries

More Complex SQL Queries

8.5.1 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 unknown (exists but is not known), value not available (exists but is purposely withheld), or value not applicable (the attribute is undefined for this tuple).
- Consider the following examples to illustrate each of the meanings of NULL.

  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 LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.
- It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish between the different meanings of NULL.

In general, 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 theresults (or truth values) of three-when the logical connectives AND, OR, and NOT are used. Table 8.1 shows the resulting values. valued logical expressions.

TABLE 8.1 LOGICAL CONNECTIVES IN THREE-VALUED LOGIC

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

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

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

- SQL allows queries that check whether an attribute value is NULL. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators IS or IS NOT. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN). Query 18 illustrates this.
- **Query 18.** Retrieve the names of all employees who do not have supervisors.

  Q18: **SELECT** Fname, Lname
  FROM EMPLOYEE
  WHERE Super_ssn **IS** NULL;

8.5.2 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. Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the 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, while 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, 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.

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

  Q4A: **SELECT DISTINCT** Pnumber
  FROM PROJECT
  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' );
```

- If a nested query returns a single attribute and a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a table (relation), which is a set or multiset of tuples.
- SQL allows the use of tuples of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:
- 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.
- In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query).
- 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.

- Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>.
- 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.
- Example: Retrieve 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);

- Notice that this query can also be specified using the MAX aggregate function (discussed later).
- In general, there are several levels of nested queries. Ambiguity among attribute names exist if attributes have the same name—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.
- For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, specify and refer to an alias (tuple variable) for that relation.

8.5.3 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.**
- Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same Gender as the employee.

```
Q16:  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.Gender=D.Gender );
```

- In the nested query of Q16, qualify E.Gender because it refers to the Gender attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Gender. If there were any unqualified references to Gender in the nested query, they would refer to the Gender attribute of DEPENDENT. However, if the attributes Fname and Ssn of EMPLOYEE appear in the nested query, there is no ambiguity because the DEPENDENT relation does not have attributes called Fname and Ssn.
- It is generally advisable to create tuple variables (aliases) for all the tables referenced in an SQL query to avoid potential errors and ambiguities, as illustrated in Q16.
- In correlated nested query, **the nested query is evaluated once for each tuple (or combination of tuples) in the outer query**.
- Example: For each EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same Gender 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.
- In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can always be expressed as a single block query. For example, Q16 may be written as in Q16A:

```
Q16A:     SELECT E.Fname, E.Lname
          FROM EMPLOYEE AS E, DEPENDENT AS D
          WHERE E.Ssn=D.Essn AND E.Gender=D.Gender
          AND E.Fname=D.Dependent_name;
```

- The **original SQL implementation on SYSTEM R also had a CONTAINS comparison operator**, which was used to compare two sets or multisets**. This operator was subsequently dropped from the language**, possibly because of the difficulty of implementing it efficiently.
- Most commercial implementations of SQL do not have this operator.

- The CONTAINS operator compares two sets of values and returns TRUE if one set contains all values in the other set. Query 3 illustrates the use of the CONTAINS operator.
- **QUERY 3** Retrieve the name of each employee who works on all the projects controlled by department number 5.

```
Q3:     SELECT FNAME, LNAME
        FROM EMPLOYEE
        WHERE ( (SELECT PNO
                FROM  WORKS_ON
                WHERE SSN=ESSN)
        CONTAINS
                (SELECT PNUMBER
                FROM PROJECT
                WHERE
                DNUM=5) );
```

- In Q3, the second nested query (which is not correlated with the outer query) retrieves the project numbers of all projects controlled by department 5.
- For each employee tuple, the first nested query (which is correlated) retrieves the project numbers on which the employee works; if these contain all projects controlled by department 5, the employee tuple is selected and the name of that employee is retrieved. Notice that the CONTAINS comparison operator has a similar function to the DIVISION operation of the relational algebra and to universal quantification in relational calculus.
- Because the CONTAINS operation is not part of SQL, use other techniques, such as the EXISTS function, to specify these types of queries, as described in Section 8.5.4.

8.5.4 The EXISTS and UNIQUE Functions in SQL

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

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

```
Q16B:     SELECT E.Fname, E.Lname
          FROM EMPLOYEE AS E
          WHERE EXISTS ( SELECT *
          FROM DEPENDENT AS D
          WHERE E.Ssn=D.Essn AND E.Gender=D.Gender
          AND E.Fname=D.Dependent_name);
```

- EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.
- In Q16B, the nested query references the Ssn, Fname, and Gender attributes of the EMPLOYEE relation from the outer query.
- For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Gender, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple.

NOT EXISTS:

- NOT EXISTS(Q) returns TRUE if there are no tuples in the result of nested query Q, and it returns FALSE otherwise.

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

```
Q6:   SELECT Fname, Lname
      FROM EMPLOYEE
      WHERE NOT EXISTS ( SELECT *
                         FROM DEPENDENT
                         WHERE Ssn=Essn );
```

- In Q6, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple.
- If none exist, the EMPLOYEE tuple is selected because the WHERE-clause condition will evaluate to TRUE in this case.
- 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 that EMPLOYEE tuple is selected to retrieve its Fname and Lname.

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

```
Q7:    SELECT Fname, Lname
       FROM EMPLOYEE
       WHERE EXISTS ( SELECT *
                      FROM DEPENDENT
                      WHERE Ssn=Essn )
             AND
             EXISTS ( SELECT *
                      FROM DEPARTMENT
                      WHERE Ssn=Mgr_ssn );
```

- Specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, select the EMPLOYEE tuple.
- **Query Q3:** Retrieve the name of each employee who works on all the projects controlled by department number 5 can be written using EXISTS and NOT EXISTS in SQL systems. Two ways of specifying this query Q3 in SQL as Q3A and Q3B.

```
Q3A:  SELECT Fname, Lname
      FROM EMPLOYEE
      WHERE NOT EXISTS ( ( SELECT Pnumber
                           FROM PROJECT
                           WHERE Dnum=5)
                           EXCEPT ( SELECT Pno
                                    FROM WORKS_ON
                                    WHERE Ssn=Essn) );
```

- In Q3A, the first subquery (which is not correlated with the outer query) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first subquery result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.

- The second option is shown as Q3B. Two-level nesting is needed in Q3B and that this formulation is quite a bit more complex than Q3A, which uses NOT EXISTS and EXCEPT.

```
Q3B:  SELECT Lname, Fname
      FROM EMPLOYEE
      WHERE NOT EXISTS ( SELECT *
                                  FROM WORKS_ON B
                                  WHERE ( B.Pno IN ( SELECT Pnumber
                                  FROM PROJECT
                                  WHERE Dnum=5 )
            AND
            NOT EXISTS ( SELECT *
                          FROM WORKS_ON C
                          WHERE C.Essn=Ssn
                          AND C.Pno=B.Pno )));
```

- In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose Pno is of a project controlled by department 5, if there is not a WORKS_ON (C) tuple with the same Pno and the same Ssn as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on.
- There is another SQL function, **UNIQUE(Q)**, which 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.

8.5.5 Explicit Sets and Renaming of Attributes 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 in SQL.
- **Query 17.** Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17: 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. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses. For example, Q8A shows how query Q8 can be slightly changed to Q8A. The new names will appear as column headers in the query result.
- Retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as Employee_name and Supervisor_name.

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;

8.5.6 Joined Tables in SQL and Outer Joins

- The concept of a joined table (or joined relation) was incorporated into SQL to permit users to specify a table resulting from a join operation in the FROM clause of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause.
- Retrieve the name and address of every employee who works for the 'Research' department.

Q1A: SELECT Fname, Lname, Address
FROM (EMPLOYEE **JOIN** DEPARTMENT **ON** Dno=Dnumber)
WHERE Dname='Research';

- The FROM clause in Q1A contains a single joined table. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT.
- The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN.
- In a NATURAL JOIN on two relations R and S, no join condition is specified.
- An implicit EQUIJOIN condition for each pair of attributes with the same name from R and S is created. Each such pair of attributes is included only once in the resulting relation.

- Different SQL JOINs
  - INNER JOIN: Returns all rows when there is at least one match in BOTH tables
  - LEFT JOIN: Return all rows from the left table, and the matched rows from the right table
  - RIGHT JOIN: Return all rows from the right table, and the matched rows from the left table
  - FULL JOIN: Return all rows when there is a match in ONE of the tables
- 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.
- This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as Dname, Dno (to match the name of the desired join attribute Dno in the EMPLOYEE table), Mssn, and Msdate. The implied join condition for this NATURAL JOIN is EMPLOYEE.Dno=DEPT.Dno, because this is the only pair of attributes with the same name after renaming:

```
Q1B: SELECT Fname, Lname, Address
        FROM (EMPLOYEE NATURAL JOIN
                (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
        WHERE Dname='Research';
```

- The default type of join in a joined table is called an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query Q8A, 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.
- In SQL, this is handled by explicitly specifying the keyword OUTER JOIN in a joined table, as illustrated in Q8B:

```
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);
```

- In SQL, the options available for specifying joined tables include:

* INNER JOIN (only pairs of tuples that match the join condition are retrieved, same as JOIN),
* LEFT OUTER JOIN (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table),
* RIGHT OUTER JOIN (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and
* The FULL OUTER JOIN (returns all rows from the left table (table1) and from the right table (table2). It combines the result of both LEFT and RIGHT joins).

- In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN).
- The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation although this should be used only with the utmost care because it generates all possible tuple combinations.
- 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.
- For example, Q2A is a different way of specifying query Q2 using the concept of a joined table:

```
Q2A: SELECT Pnumber, Dnum, Lname, Address, Bdate
         FROM ((PROJECT JOIN DEPARTMENT ON Dnum=Dnumber)
                 JOIN EMPLOYEE ON Mgr_ssn=Ssn)
         WHERE Plocation='Stafford';
```

- Not all SQL implementations have implemented the new syntax of joined tables.
- In some systems, a different syntax was used to specify outer joins by using the **comparison operators +=, =+, and +=+ for left, right, and full outer join, respectively**, when specifying the join condition. For example, this syntax is available in Oracle.
- To specify the left outer join in Q8B using this syntax, write the query Q8C as follows:

```
Q8C: SELECT E.Lname, S.Lname
         FROM EMPLOYEE E, EMPLOYEE S
```

WHERE E.Super_ssn += S.Ssn;

8.5.7 Aggregate Functions in SQL

- **Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary.
- Grouping is used to create subgroups of tuples before summarization.
- 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 SELECT clause or in a HAVING clause.
- 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. (Note: Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, DATE, TIME, and TIMESTAMP domains have total orderings on their values, as do alphabetic strings.)
- **Query 19.** Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

  Q19: SELECT SUM (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)

  FROM EMPLOYEE;
- **Query 20.** 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.

  Q20: SELECT SUM (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)

  FROM (EMPLOYEE **JOIN** DEPARTMENT **ON** Dno=Dnumber)

  WHERE Dname='Research';
- **Queries 21 and 22**. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

  Q21: SELECT COUNT (*)

  FROM EMPLOYEE;

  Q22: SELECT COUNT (*)

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

- Here the asterisk (*) refers to the rows (tuples), so COUNT (*) returns the number of rows in the result of the query. The COUNT function is used to count values in a column rather than tuples, as in the next example.
- **Query 23.** Count the number of distinct salary values in the database.

```
    Q23: SELECT COUNT (DISTINCT Salary)
        FROM EMPLOYEE;
```

- Duplicate values will not be eliminated if COUNT(SALARY) is used instead of COUNT(DISTINCT SALARY) in Q23. However, any tuples with NULL for SALARY will not be counted.
- In general, NULL values are discarded when aggregate functions are applied to a particular column (attribute).
- The preceding examples summarize a whole relation (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce single tuples or single values.
- They illustrate how functions are **applied to retrieve a summary value or summary tuple** from the database.
- These functions can also be **used in selection conditions** involving nested queries.
- **Query 5:** To retrieve the names of all employees who have two or more dependents.

```
    Q5: SELECT Lname, Fname
        FROM EMPLOYEE
        WHERE ( SELECT COUNT (*)
                FROM DEPENDENT
                WHERE Ssn=Essn ) >= 2;
```

- The correlated nested query counts the number of dependents that each employee has. If this is greater than or equal to two, the employee tuple is selected.

8.5.8 Grouping: The GROUP BY and HAVING Clauses

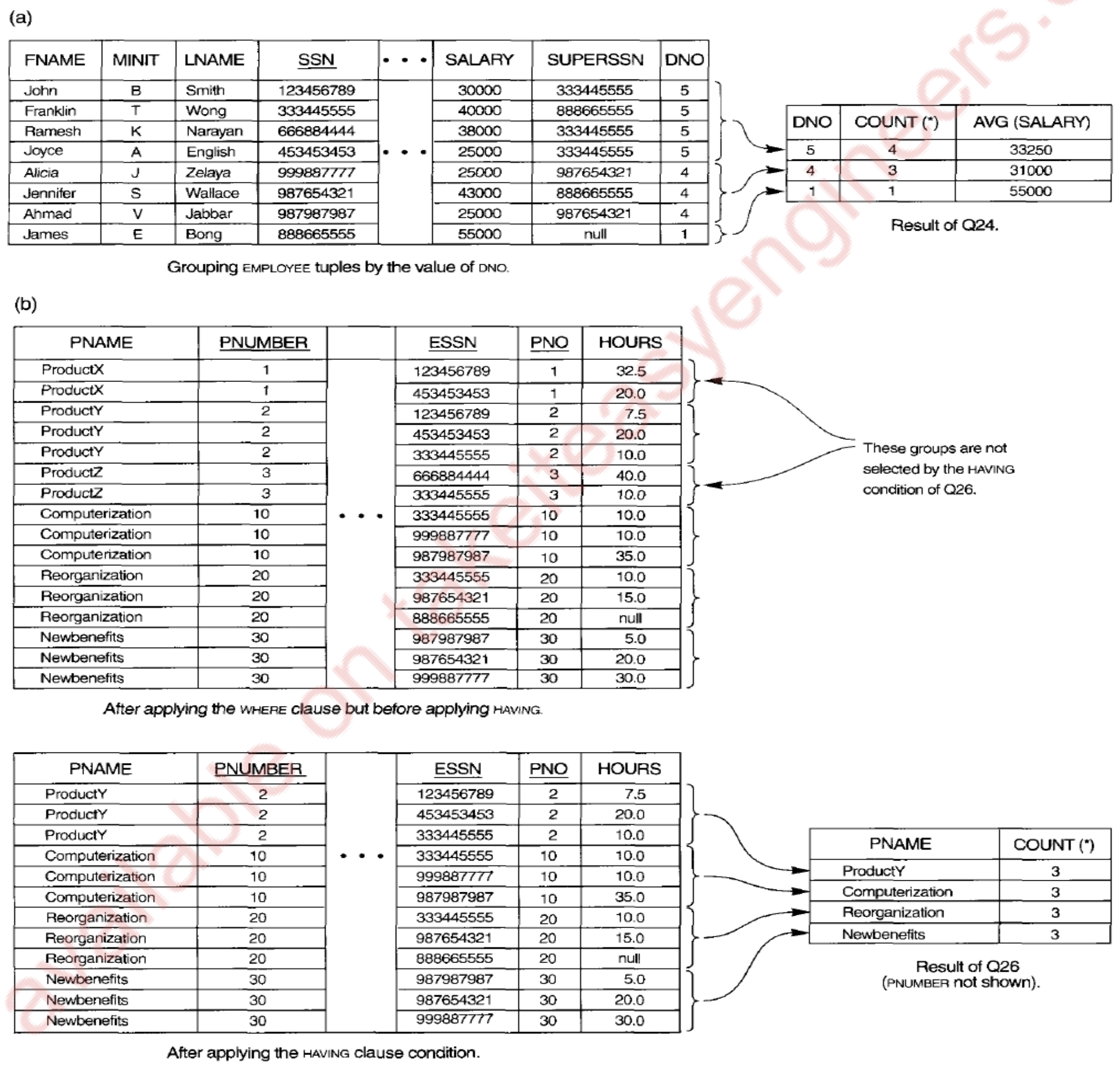
- The aggregate functions are applied to subgroups of tuples in a relation, where the subgroups are based on some attribute values. For example, to find the average salary of employees in each department or the number of employees who work on each project.

- In these cases, partition the relation into nonoverlapping 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).
- Then apply the function to each such group independently to produce summary information about each group.
- 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).
- **Query 24.** For each department, retrieve the department number, the number of employees in the department, and their average salary.

  Q24: SELECT Dno, **COUNT** (*), **AVG** (Salary)
  FROM EMPLOYEE
  GROUP BY Dno;
- In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute Dno. Hence, each group contains the employees who work in the same department.
- The COUNT and AVG functions are applied to each such group of tuples.
- Notice that the SELECT clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples.
- Figure 8.6(a) illustrates how grouping works on Q24; it also shows the 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 Q24.
- **Query 25.** For each project, retrieve the project number, the project name, and the number of employees who work on that project.

  Q25: SELECT Pnumber, Pname, **COUNT (*)**
  FROM PROJECT**,** WORKS_ON
  WHERE Pnumber=Pno
  GROUP BY Pnumber, Pname;

- Q25 shows how a join condition is used in conjunction with GROUP BY. In this case, the grouping and functions are applied after the joining of the two relations.
- To retrieve the values of these functions only for groups that satisfy certain conditions, use HAVING caluse.
- SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose. 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. This is illustrated by Query 26.



(a)

FNAME	MINIT	LNAME	SSN	• • •	SALARY	SUPERSSN	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.

(b)

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).

FIGURE 8.6 Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

- **Query 26**. 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.

Q26: SELECT Pnumber, Pname, **COUNT** (*)

FROM PROJECT, WORKS_ON

WHERE Pnumber=Pno

GROUP BY Pnumber, Pname

HAVING COUNT (*) > 2;

- Notice that while selection conditions in the WHERE clause limit the tuples to which functions are applied, the HAVING clause serves to choose whole groups. Figure 8.6(b) illustrates the use of HAVING and displays the result of Q26.
- **Query 27.** For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

Q27: SELECT Pnumber, Pname, **COUNT** (*)

FROM PROJECT, WORKS_ON, EMPLOYEE

WHERE Pnumber=Pno **AND** Ssn=Essn **AND** Dno=5

GROUP BY Pnumber, Pname;

- To count the total number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work.
- Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. The following shows a **incorrect query**:

SELECT Dname, **COUNT** (*)

FROM DEPARTMENT, EMPLOYEE

WHERE Dnumber=Dno **AND** Salary>40000

GROUP BY Dname

HAVING COUNT (*) > 5;

- This is incorrect because it will select only departments that have more than five employees who each earn more than $40,000. The rule is that the **WHERE clause is executed first**, **to select individual tuples or joined tuples**; the HAVING clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than $40,000 before the function in the HAVING clause is applied.
- One way to write this query correctly is to use a nested query, as shown in Query 28.
- **Query 28**. 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.

Q28: SELECT Dnumber, **COUNT** (*)

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

8.5.9 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.

- The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause.
- Finally, **ORDER BY** specifies an **order for displaying the result of a query**.
- In order to formulate queries correctly, it is useful to consider the steps that define the meaning or semantics of each 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**. Conceptually, **ORDER BY is applied at the end** to sort the query result.
- Of course, this is not an efficient way to implement the query in a real system, and 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 operator.
- Some users may be more comfortable with one approach, whereas others may be more comfortable with another. 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**.

- Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional **burden on the user is to determine which of the alternative specifications is the most efficient to execute**.
- Ideally, the user should worry only about specifying the query correctly, whereas the DBMS would determine how to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others.

8.6 Specifying Constraints as Assertions and Triggers

- 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.
- **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, 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. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.

- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.
- The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition*. 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. In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.
- The CHECK clause and constraint condition can also be used to specify constraints on *individual* attributes and domains and on *individual* tuples .
- A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated*. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases.
- The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*.
- On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.
- Another important statement in SQL is CREATE TRIGGER. 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. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation.
- A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user.
- The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates.
- The CREATE TRIGGER statement is used to implement such actions in SQL.

8.8 Views (Virtual Tables) in SQL

8.8.1 Concept of A View in SQL

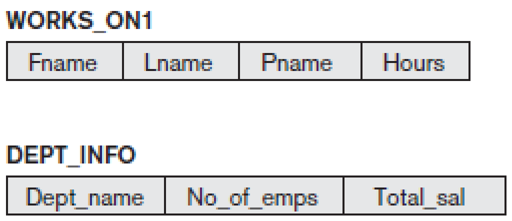
- A **view** in SQL terminology is a single table that is derived from other tables.
- These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database.
- This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.
- A view is used to specify a table that is referenced frequently, even though it may not exist physically.
- **Example:-** Referring to the COMPANY database, one 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, one can define a view that is specified as the result of these joins. Then queries on view can be issued, which are specified as single table retrievals rather than as retrievals involving two joins on three tables.
- The EMPLOYEE, WORKS_ON, and PROJECT tables are called as the **defining tables** of the view.

8.8.2 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, no need 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.
- The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 8.7 when applied to a sample database schema.

V1: CREATE VIEW WORKS_ON1

```
         AS SELECT Fname, Lname, Pname, Hours
                FROM EMPLOYEE, PROJECT, WORKS_ON
              WHERE Ssn=Essn AND Pno=Pnumber;
      V2: 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;
```


WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

Figure 8.7 Two views specified on the sample database schema

- In V1, new attribute names are not specified for the view WORKS_ON1. It *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.
- View V2 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.
- SQL queries are specified on a view—or virtual table—in the same way, queries are specified involving base tables.
- **Example:-** To retrieve the last name and first name of all employees who work on the 'ProductX' project, utilize the WORKS_ON1 view and specify the query as in QV1:

```
QV1: 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 the tuples are modified 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 *a query is specified* 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 a view is not needed any more, use the **DROP VIEW** command to dispose it.
- **Example**:- To get rid of the view V1, use the SQL statement in V1A:

  V1A: DROP VIEW WORKS_ON1;

8.8.3 View Implementation and View Update

- 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.
- **Example:-** The query QV1 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 the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY' is issued. This view update is shown in UV1:

```
UV1: UPDATE WORKS_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.
- **Example:-** Here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(a): UPDATE  WORKS_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): UPDATE PROJECT **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.
- **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: UPDATE DEPT_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, a certain procedure is required for choosing one of the possible updates as the most likely one.
- Summarizing, the following observations can be made:
  - 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.

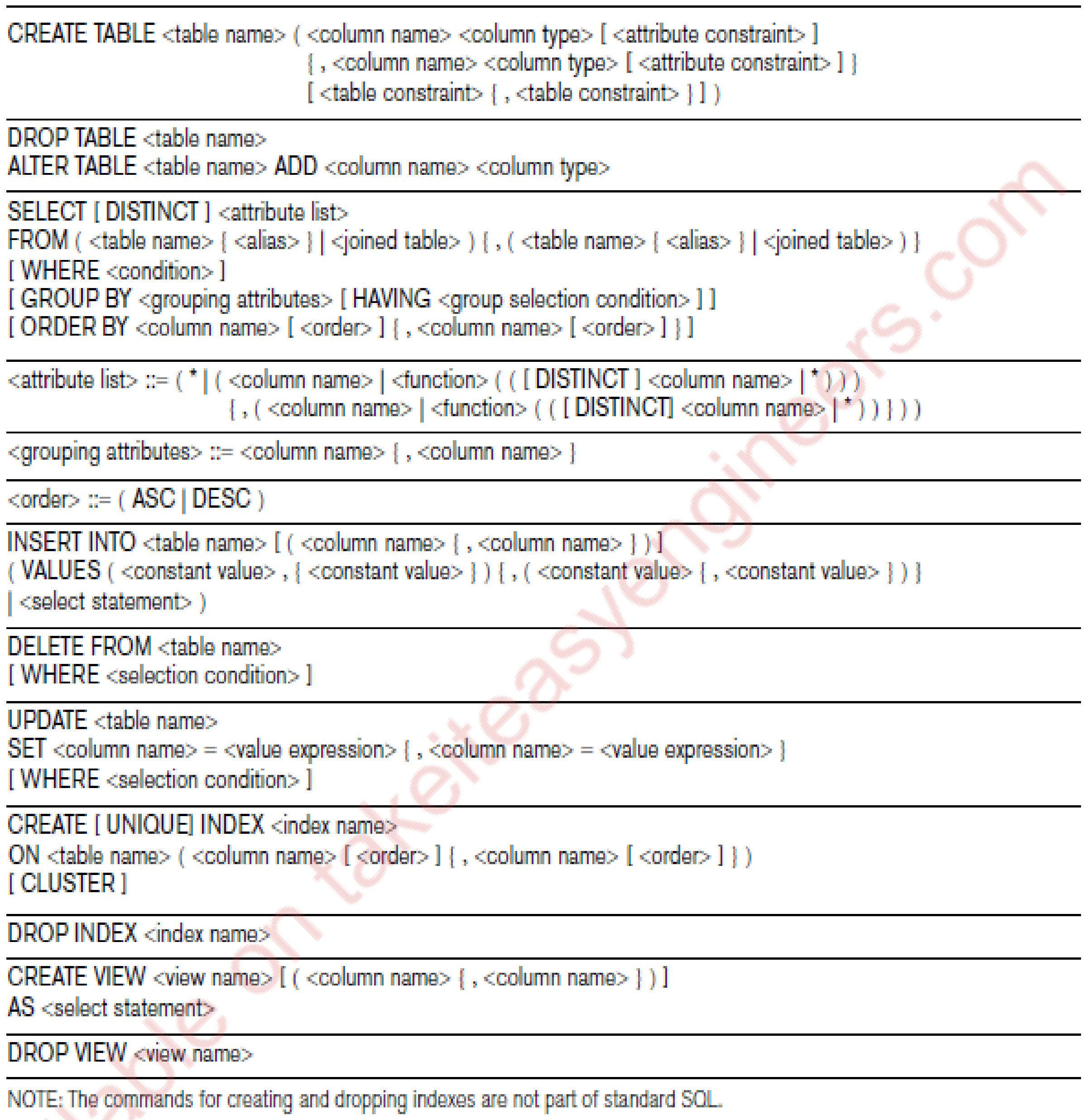
- In SQL, the clause **WITH CHECK OPTION** must be added at the end of the view definition if a view is to be updated. This allows the system to check for view updatability and to plan an execution strategy for view updates.

8.9 Additional Features of SQL

- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Language Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules).
- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. These commands are defined in a storage definition language (SDL). Earlier versions of SQL had commands for **creating indexes**, but these were removed from the language because they were not at the conceptual schema level.
- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.
- SQL has language constructs for specifying the granting and revoking of privileges to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or
- UPDATE—to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called **GRANT** and **REVOKE**.
- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational**. Capabilities such as creating complex-structured attributes (also called nested relations),

specifying abstract data types (called UDTs or user-defined types) for attributes and tables, creating object identifiers for referencing tuples, and specifying operations on types.

- SQL and relational databases can interact with new technologies such as XML and OLAP.



```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                          { , <column name> <column type> [ <attribute constraint> ] }
                          [ <table constraint> { , <table constraint> } ] )

DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>

SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]

<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                     { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )

<grouping attributes> ::= <column name> { , <column name> }

<order> ::= ( ASC | DESC )

INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )

DELETE FROM <table name>
[ WHERE <selection condition> ]

UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[ WHERE <selection condition> ]

CREATE [ UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Table 8.2 Summary of SQL Syntax

DATABASE APPLICATION DEVELOPMENT

6.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we learn how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called Embedded SQL. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

6.1.1 Embedded SQL

Embedding SQL commands in a host language program is straightforward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL.

In particular, some special host language variables *must* be declared in SQL There are, however, **two complications** to bear in mind.

First, the data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by **casting data values** appropriately before passing them to or frorn SQL commands

The second complication h~s to do with SQL being set-oriented,
and is addressed using cursors

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such hos tlanguage
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 how they would look in a C program and, as usual in C. are separated by semicolons

For example. we can declare variables *c-sname*, *c_sid*, *c_mt'ing*, and *cage* (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 *csid;*
short *crating;*
float *cage;*
EXEC SQL END DECLARE SECTION

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_snamc* has the type CHARACTER(20) when referred to in an SQL statement, *csid* has the type INTEGER, *crating* has the type SMALLINT, and *cage* 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.

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 me based on the values of the host language variables contained in it, into the Sailors relation:

EXEC SQL
INSERT INTO Sailors VALUES *(:c_sname, :csid, :crating, :cage);*
Observe that a semicolon terminates the command, as per the convention for terminating statements in C.

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

6.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an ***impedance mismatch*** occurs because SQL operates on *set"* 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 (because every query returns a set of rows). Once a it is declared, we can **open** it (which 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 to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially 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 JWS computed by an Embedded SQL statement: usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.

INSERT, DELETE, and UPDATE staternents typically require no cursor, although
some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning
a value to the host variable c_sid, declared earlier, as follows:

```
EXEC SQL SELECT
INTO
FROM
WHERE
S.sname, S.age
:c_sname, :c_age
Sailors S
S.sid = :c_sid;
```

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

the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable *cminmting*

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the cOlnmand with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_minrating;
```

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursomame [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, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE). If it is IIpdatable, 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 *sinfa* 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;
```

This Embedded SQL statement modifies the *rating* value of the row currently pointed to by cursor *sinfa;* similarly, we can delete this row by executing the next statement:

```
DELETE Sailors S
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 conunitted. 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. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

6.1.3 Dynamic SQL

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, even though there is (presumably) some algorithm by which the application can construct the necessary SQL statements once a user's command is issued. SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL.

We illustrate the two main commands, PREPARE and EXECUTE, through a simple 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*. (Since *readytogo* is an SQL variable, just like a cursor name, it is not prefixed by a colon.)

The third statement executes the command. Many situations require the use of Dynamic SQL. However, note that the preparation of a Dynamic SQL command occurs at run-time and is run-time overhead. Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executecl as often as desired. Consequently you should limit the use of Dynamic SQL to situations in which it is essential.

6.2 AN INTRODUCTION TO JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. The details of 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.

Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API). In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *'Without recompilation.* Thus, 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. One interesting point to note is that 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**. Therefore, in the remainder of this section, we refer to a data storage subsystem with which a driver interacts as a data source.

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, and 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 opcn, 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.

JDBC drivers are available for all major database sytems.

6.2.1 Architecture

The architecture of JDBC has four main components:

- the *application,*
- the*driver manager,*
- several data source specific *dr-iveTs,*
- and the corresponding
- *data SOURces.*

The *application* initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API. The primary goal of the *dr-iver manager* is to load JDBC drivers and pass JDBC function calls from the application to the correct driver. The driver manager also handles JDBC initialization and information calls from the applications and can log all function calls. In addition, the driver manager performs· some rudimentary error checking. The *dr-iver* establishes the connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard.

The *data source* processes commands from the driver and returns the results.Depending on the relative location of the data source and the application, several architectural scenarios are possible. Drivers in JDBC are cla.ssified 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. Bridges have the advantage that it is easy to piggyback the applica.tion 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. In addition, the user is limited to the functionality that the ODBC driver supports.

Type II Direct Thanslation 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. usually ,vritten using a combination of C++ and Java; it is dynamically linked and specific to the data source.

This architecture performs significantly better than a JDBC-ODBC bridge. One disadvantage is that 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 (Le., thenetwork bridge) is not DBMS-specific. The JDBC driver loaded by the ap~ plication 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.

6.3 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from prograrl1s 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 cla.sses and interfaces are part of the java. sql package.
.

6.3.1 JDBC Driver Management

In .lDBe, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers. The Drivermanager cla.ss 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. This is accomplished by using the Java mechanism for dynamically loading classes. The static method forName in the Class class returns the Java class as specified in the argument string and executes its static constructor. The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

The following Java example code explicitly loads a JDBC driver: Class.forName("oracle/jdbc.driver.OracleDriver");

There are two other ways of registering 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 thismethod 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.

6.3.2 Connections

A session with a data source is started through creation of a Connection object;A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the samedata source.

Connections are specified through a **JDBC** URL, a URL that uses the jdbc protocol. Such a URL has the form

jdbc:<subprotocol>:<otherParameters>

For example,
The Connection class has methods to set the

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

JDBC Connections: Remember to close connections to data sources and return shared connections to the connection pool. Database systems have a limited number of resources available for connections, and orphan connections can often only be detected through time-outs- and while the database system is waiting for the connection to time-out, the resources
used by the orphan connection are wasted. 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 SQLExceptionand
public void setTransactionlsolation(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 *1* can be
set as follows:

- TRANSACTIONJNONE
- TRANSACTIONJREAD.UNCOMMITTED
- TRANSACTIONJREAD.COMMITTED
- TRANSACTIONJREPEATABLEJREAD
- TRANSACTION.BERIALIZABLE

• 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. we already discussed these two functions. Establishing a connection to a data source is a costly operation since it involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory.

In case an application establishes many different connections from different parties (suchas 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. Connection pooling can be handled either by specialized code in the application, or the optional j avax. sql package, which provides functionality for connection pooling and allows us to set different parameters.

6.3.3 Executing SQL Statements

We now discuss how to create and execute SQL statements using JDBC.
. JDBC supports three different ways of executing statements:

- Statement,
- PreparedStatement,
- CallableStatement.

The Statement

This class is the base class for the other two statment classes. It allows us to query the data source with any static or dynamically generated SQL query.

The PreparedStatement cla,Cis dynamicaJly 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 (representing the SQL statement) is created.

Consider the sample code using a PreparedStatment object The SQL query specifies the query string, but uses "1' for the values of the parameters, which are set later using methods setString, setFloat, and setlnt. The "1' 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 staternents, as in Figure 6.3. The method setString is one way

```
/ / 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();
```

Figure 6.3 SQL Update Using a PreparedStatement Object to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters 0 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 integerindicating the number of rows the SQL statement modified; it returns 0 forsuccessful execution without modifying any rows.

The executeQuery method is used if the SQL statement returns data, such &"l in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object, which we discuss next. The execute method is more general than executeQuery and executeUpdate; the references at the end of the chapter provide pointers with more details.

6.3.4 Resul,tSets

the statement executeQuery returns a,ResultSet object, which is similar to a cursor. 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 0 method. The next method returns false if there are no more rows in the query answer, and true other\vise. The code fragment shown in Figure 6.4

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
}
```

Figure 6.4 Using a ResultSet Object

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 0 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 0 moves to the first row, and last 0 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. Figure 6.5 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.

I SQL Type I Java cla.c;;s I ResultSet get method I

BIT Boolean getBooleanO

CHAR String getStringO

VARCHAR String getStringO

DOUBLE Double getDoubleO

FLOAT Double getDoubleO

INTEGER Integer getIntO

REAL Double getFloatO

DATE java.sql.Date getDateO

TIME java.sql.Time getTimeO

TIMESTAMP java.sql.TimeStamp getTimestamp ()

Figure 6.5 Reading SQL Datatypes from a ResultSet Object

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

6.3.5 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 0 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 0 returns an SQLState identifier based on the SQL:1999 specification, as discussed in Section 6.1.1... public i:p.t getErrorCode () retrieves a vendor-specific error codeIII public SQLException getNextExceptionO gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQL Warning is a subclass of SQLException. Warnings are not H•.'3 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 a., part of the try''-catch block around a java. sql statement.

Typical code for obtaining SQLWarnings looks similar to the code shown inFigure 6.6.

```
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
```

Figure 6.6 Processing JDBC Warnings and Exceptions

6.3.6 Examining Database Metadata

We can use tlw DatabaseMetaData object to obtain information about the database system itself, as well as information frorn 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); we list some methods here:

• public ResultSet getCatalogs 0 throws SqLException. This function
returns a ResultSet that can be used to iterate over all the catalog relations.
The functions getIndexInfo 0 and getTables 0 work analogously.

• pUblic int getMaxConnections 0 throws SqLException. This function
returns the ma.ximum number of connections possible.

example code fragment that examines all database metadata shown in Figure 6.7.

```
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") +" ");
}
}
```

Figure 6.7 Obtaining Infon-nation about it Data Source

6.4 SQLJ

SQLJ (short for 'SQL-Java') was developed by the SQLJ Group, a group of database vendors and Sun. SQLJ was developed to complement the dynamic way of creating queries in JDBC with a static model. It is therefore very close to Embedded SQL. Unlike JDBC, having semi-static SQL queries allows the compiler to perform SQL syntax checks, strong type checks of the compatibility of the host variables with the respective SQL attributes, and consistency of the query with the database schema-tables, attributes, views, and stored procedures--all at compilation time.

For example, in both SQLJ and Embedded SQL, variables in the host language always are bound statically to the same arguments, whereas in JDBC, we need separate statements to bind each variable to an argument and to retrieve the result. For example, the following SQLJ statement binds host language variables title, price, and author to the
return values of the cursor books.

```
#sql books = {
SELECT title, price INTO :title, :price
FROM Books WHERE author = :author
};
```

In JDBC, we can dynamically decide which host language variables will hold
the query result. In the following example, we read the title of the book into
variable ftitle if the book was written by Feynman, and into variable otitle
otherwise:

```
/ / assume we have a ResultSet cursor rs
author = rs.getString(3);
if (author=="Feynman") {
ftitle = rs.getString(2):
}
else {
otitle = rs.getString(2);
}
```

when writing SQLJ applications, we just write regular Java code and embed SQL statements according to a set of rules. SQLJ applications are pre-processed through an SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ Java library. The modified program code can then be compiled by any Java compiler. Usually the SQLJ Java library makes calls to a JDBC driver, which handles the connection to the datab&'3e system.

An important philosophical difference exists between Embedded SQL and SQLJ and JDBC. Since vendors provide their own proprietary versions of SQL, it is advisable to write SQL queries according to the SQL-92 or SQL:1999 standard. However, when using Embedded SQL, it is tempting to use vendor-specific SQL constructs that offer functionality beyond the SQL-92 or SQL:1999 standards. SQLJ and JDBC force adherence to the standards, and the resulting code is much more portable across different database systems. In the remainder of this section, we give a short introduction to SQLJ.

6.4.1 Writing SQLJ Code

We will introduce SQLJ by means of examples. Let us start with an SQLJ code fragment that selects records from the Books table that match a given author.

```
String title; Float price; String atithor;
#sql iterator Books (String title, Float price);
Books books;
/ / the application sets the author
/ / execute the query and open the cursor
#sql books = {
SELECT title, price INTO :titIe, :price
FROM Books WHERE author = :author
};
/ / retrieve results
while (books.next()) {
System.out.println(books.titleO + ", " + books.price());
}
books.close() ;
```

The corresponding JDBC code fragment looks as follows (assuming we also declared price, name, and author:

```
PrcparcdStatcment stmt = connection.prepareStatement(
" SELECT title, price FROM Books WHERE author = ?");
/ / set the parameter in the query ancl execute it
stmt.setString( 1, author);
```

ResultSet 1'8 = stmt.executeQuery();

/ / retrieve the results

while (rs.next()) {

208 CHAPTER

System.out.println(rs.getString(l) + ", " + rs.getFloat(2));

}

Comparing the JDBC and SQLJ code, we see that the SQLJ code is much easier to read than the JDBC code. Thus, SQLJ reduces software development and maintenance costs. Let us consider the individual components of the SQLJ code in more detail. 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 goesthrough five steps:

• 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.

• Instantiate an Iterator Object from the New Iterator Class: We instantiated our iterator in the statement Books books;.

• Initialize the Iterator Using a SQL Statement: In our example, this happens through the statement #sql books ;;;;;; ....

• Iteratively, Read the Rows From the Iterator Object: This step is very similar to reading rows through a ResultSet object in JDBC.

• Close the Iterator Object.

There are two types of iterator classes: named iterators and 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 as
in our previous example where we could retrieve the title colunm from the Books
table using the expression books. titIe ().

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 eonstruct, similar to Embedded SQL. Both iterator types have the same performance; which iterator to use depends on the programmer's taste. Let us revisit our example. \Ve 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 3,.'3 follows:

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

6.5 STORED PROCEDURES

It is often important to execute some parts of the application logic directly in the process space of the database system. Running application logic directly at the databa.se has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized, while at the same time utilizing the full power of the databa.se server. When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application.

If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor. In contrast, a stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at allL Stored procedures are also beneficial for software engineering rea,sons.

Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance ea."lY.

In addition, application programmers do not need to know the the databa.se schema if we encapsulate all databa.'3e access into stored procedures. Although they,are called stored *procedur'es*, they do not have to be procedures in a programming language sense; they can be functions.

6.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure(i.S. vVe see that stored procedures must have a name; this stored procedurehas the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQLstatement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
FROM Customers C, Orders a
WHERE C.cid = O.cid
GROUP BY C.cid, C.cname
```

Figure 6.8 A Stored Procedure in SQL

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. Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR. Let us look at an example of a stored procedure with arguments.

The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

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

Figure 6.9 A Stored Procedure with Arguments

Stored procedures do not have to be written in SQL; they can be written in any host language. As an example, the stored procedure shown in Figure 0.10 is a Java function that is dynamicallyexecuted by the databa.<. ;e server whenever itis called by the dient:

6.5.2 Calling Stored Procedures

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

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:/ / /c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

```
CALL storedProcedureName(argumentl, argument2, ... , argumentN);
```

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[lO];
long qty;
```

EXEC SQL END DECLARE SECTION
/ / set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL staternents 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=
COIl. prepareCall(" {call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
while (rs.next())

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:
/ / create the cursor class
#sql !terator CustomerInfo(int cid, String cname, int count);
/ / create the cursor
CustomerInfo customerinfo;
/ / call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.nextO) {
System.out.println(customerinfo.cid() + "," +
customerinfo.count()) ;
}

6.5.3 SQLIPSM

All major databa...<;e systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL. In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendo rspecific
languages.

In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations. 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 sqIDataType

local variable declarations
function code;

Each parameter is a triple consisting of the the parameter name, and the SQL datatype of the parameter. We can seen very simple SQL/PSM procedures in Section6.5.1. In this case, the local variable declarations were empty, and the procedure code consisted of an SQL query.

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her *cid* and a year. The function returns the rating of the customer, which is defined a...'3 follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purcha...<;ed between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

CREATE PROCEDURE RateCustomer
Database Appl'ication Development
(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;

Let us use this example to give a short overview of some SQL/PSM constructs:

• 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'.

• vVe 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 h&<; 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

staternents:

END LOOP

• Queries can be used as part of expressions in branches; queries that return a single ;ralue can be assigned to variables as in our example above.

• 'We can use the same cursor statements &s in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do nothave to be prefixed by a colon ':'.

We only gave a very short overview of SQL/PSM; the references at the end of the chapter provide more information.

6.6 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes finished logical database design, as discussed in Section 3.8, and now consider the queries that they have to support. They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code. Recall that DBDudes settled on the following schema:

Books(*isbn:* CHAR(10), *title:* CHAR(8), *author:* CHAR(80),
qty_in_stock: INTEGER, *price:* REAL, *year_published:* INTEGER)
Customers(*cid:* INTEGER, *cname:* CHAR(80), *address:* CHAR(200))
Orders(*ordernum:* INTEGER, *isbn:* CHAR(lO), *cid:* INTEGER,
cardnum: CHAR(l6), *qty:* INTEGER, *order_date:* DATE, *ship_date:* DATE)

Now, DBDudes considers the types of queries and updates that will arise. They first create a list of tasks that will be performed in the application. Tasks performed by customers include the following. II Customers search books by author name, title, or ISBN. .. Customers register with the website. Registered customers might want to change their contact information. DBDudes realize that they have to augment the Customers table with additional information to capture login and password information for each customer; we do not discuss this aspect any further.

```
CREATE PROCEDURE SearchByISBN (IN book.isbn CHAR (10) )
SELECT B.title, B.author, B.qty_in~'3tock, B.price, B.yeaLpublished
FROM Books B
WHERE B.isbn = book.isbn
```

Placing an order involves inserting one or more records into the Orders table. Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array. To finalize the order, they write the following JDBC code shown in Figure 6.11, which inserts the elements from the array into the Orders table. Note that this code fragment assumes several Java variables have been set beforehand.

```
String sql = "INSERT INTO Orders VALUES(7, 7, 7, 7, 7, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
con.setAutoCommit(false);
try {
/ / orderList is a vector of Order objects
/ / ordernum is the current order number
/ / dd is the ID of the customer, cardnum is the credit card number
for (int i=O; iiorderList.lengthO; i++)
/ / now instantiate the parameters with values
Order currentOrder = orderList[i];
pstmt.clearParameters() ;
pstmt.setInt(l, ordernum);
pstmt.setString(2, Order.getlsbnO);
pstmt.setInt(3, dd);
pstmt.setString(4, creditCardNum);
pstmt.setlnt(5, Order.getQtyO);
pstmt.setDate(6, null);
pstmt.executeUpdate();
}
con.commit();
catch (SqLException e){
con.rollbackO;
System.out. println (e.getMessage());
}
```

Figure 6.11 Inserting a Completed Order into the Database

DBDudes writes other JDBC code and stored procedures for all of the remaining tasks. They use code similar to some of the fragments that we have seen in this chapter.

DBDudcs takes care to make the application robust by processing exceptions and warnings, as shown in Figure 6.6. DBDudes also decide to write a trigger, which is shown in Figure 6.12. Whenever a new order is entered into the Orders table, it is inserted with ship~date set to NULL. The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'. This stored procedure (whose code is not shown here) updates the (anticipated) ship_date of the new order to 'tomorrow', in case qty jlLstock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedme sets the ship_date to two weeks.

```
CREATE TRIGGER update_ShipDate
AFTER INSERT ON Orders
FOR EACH ROW
BEGIN CALL UpdatcShipDate(new); END
1* Event *j
1* Action *j
```