INDEX

Syllabus

UNIT II : Query Languages (15 Lectures)
(a) **Relational Algebra** : select and projection, Set operations like union, intersection, difference, cross product, Joins – conditional, equi join and natural joins, division, examples. Overview of relational Calculus.
(b) **Creating and altering tables**: Conversion of ER to relations with and without constraints; CREATE statement with constraints like KEY, CHECK, DEFAULT, ALTER and DROP statement.
(c) **Handling data using SQL**: selecting data using SELECT statement, FROM clause, WHERE clause, HAVING clause, ORDER BY, GROUP BY,DISTINCT and ALL predicates, Adding data with INSERT statement, changing data with UPDATE statement, removing data with DELETE statement
(d**) Functions**: Aggregate functions-AVG, SUM, MIN, MAX and COUNT, Date functions-DATEADD(),DATEDIFF(),GETDATE(),DATENAME()YEAR,MONTH, WEEK, DAY, String functions-LOWER(), UPPER(), TRIM(), RTRIM(),PATINDEX(), REPLICATE(), REVERSE(),RIGHT(), LEFT()
(e) **Joining tables**: Inner, outer and cross joins, union.
(f) **Sub queries**: sub queries with IN, EXISTS, sub queries restrictions, Nested sub queries, correlated sub queries, queries with modified comparison operations, SELECT INTO operation, UNION operation. Sub queries in the HAVING clause.

CHAPTER 1: RELATIONAL ALGEBRA

Topic Covered:

- select and projection, Set operations like union, intersection, difference, cross product, Joins – conditional, equi join and natural joins, division, examples. Overview of relational Calculus.

Q. Explain selection and projection?

Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project*

columns (π). These operations allow us to manipulate data in a single relation. Consider

the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression

σ *rating*>8(*S*2)

evaluates to the relation shown in Figure 4.4. The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 4.4 σ *rating*>8(*S*2)

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Figure 4.5 π *sname;rating*(*S*2)

The selection operator σ specifies the tuples to retain through a *selection condition*. In general, the selection condition is a Boolean combination (i.e., an expression using the logical connectives ^ and *V*) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <,<=,=, ≠=,>=, or >. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance.

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using π. The expression π *sname; rating*(*S*2) evaluates to the relation shown in Figure 4.5. The subscript *sname, rating* specifies the fields to be retained; the other fields are `projected out.' The schema of the result of a projection is determined by the fields that are projected in the obvious way.

Suppose that we wanted to find out only the ages of sailors. The expression

π *age*(*S*2)

evaluates to the relation shown in Figure 4.6. The important point to note is that although three sailors are aged 35, a single tuple with *age=35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples*, leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

Since the result of a relational algebra expression is always a relation, we can substitute an expression wherever a relation is expected. For example, we can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

π *sname;rating*(π *rating*>8(*S*2))

produces the result shown in Figure 4.7. It is obtained by applying the selection to *S*2 (to get the relation shown in Figure 4.4) and then applying the projection.

age
35.0
55.5

Figure 4.6 π *age*(*S*2)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

Figure 4.7 π *sname;rating*(π *rating*>8(*S*2))

Q. write a note on set operations?

- **Set Operations**
  The following standard operations on sets are also available in relational algebra: *union* (*U*), *intersection* (∩), *set-difference* (−), and *cross-product* (*X*).
- **Union:** *R US* returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *union compatible*, and the schema of the result is defined to be identical to the schema of *R*.
  Two relation instances are said to be **union-compatible** if the following conditions hold:
  - they have the same number of the fields, and
  - corresponding fields, taken in order from left to right, have the same *domains*.
- Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of *R U S* inherit names from *R*, if the fields of *R* have names. (This assumption is implicit in defining the schema of *R U S* to be identical to the schema of *R*, as stated earlier.)
- **Intersection:** *RUS* returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
- **Set-difference:** *R−S* returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
- **Cross-product:** *RXS* returns a relation instance whose schema contains all the fields of *R* (in the same order as they appear in *R*) followed by all the fields of *S* (in the same order as they appear in *S*). The result of *R X S* contains one tuple *hr; si* (the concatenation of tuples *r* and *s*) for each pair of tuples *r* ∈*R;*
- *s* ∈ *S*. The cross-product opertion is sometimes called **Cartesian product**.
- We will use the convention that the fields of *R X S* inherit names from the corresponding fields of *R* and *S*. It is possible for both *R* and *S* to contain one or more fields having the same name; this situation creates a *naming conflict*. The corresponding fields in *R X S* are unnamed and are referred to solely by position. In the preceding definitions, note that each operator can be applied to relation instances that are computed using a relational algebra (sub) expression.
- We now illustrate these definitions through several examples. The union of *S*1 and *S*2 is shown in Figure 4.8. Fields are listed in order; field names are also inherited from *S*1. *S*2 has the same field names, of course, since it is also an instance of Sailors. In general, fields of *S*2 may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both *S*1 and *S*2 appear only once in *S*1 *US*2. Also, *S*1 *U R*1 is not a valid operation because the two relations are not union-compatible. The intersection of *S*1 and *S*2 is shown in Figure 4.9, and the set-difference *S*1 − *S*2 is shown in Figure 4.10.

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

Fig S1U S2

Fig s1 ∩ s2

sid	sname	rating	age
22	dustin	7	45.0

sid	sname	rating	age
22	dustin	7	45.0

Fig s1 - s2

Q. Explain joins- condition, equi, natural?

- **Joins**
  The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Further, the result of a cross-product is typically much larger than the result of a join, and it is very important to recognize joins and implement them without materializing the underlying cross-product (by applying the selections and projections `on-the-fly'). For these reasons, joins have received a lot of attention, and there are several variants of the join operation.1
- **Condition Joins**
  The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:
  $R \bowtie_c S = \sigma_c(R \times S)$
- Thus ⋈ is defined to be a cross-product followed by a selection. Note that the condition *c* can (and typically *does*) refer to attributes of both *R* and *S*. The reference to an 1There are several variants of joins that are not discussed in this chapter. An important class of joins called *outer joins* is discussed in Chapter 5. attribute of a relation, say *R*, can be by position (of the form *R:i*) or by name (of the form *R:name*). As an example, the result of *S*1 ⋈*S*1*:sid*<*R*1*:sid R*1 is shown in Figure 4.12. Because *sid* appears in both *S*1 and *R*1, the corresponding fields in the result of the cross-product *S*1 *XR*1 (and therefore in the result of *S*1 ⋈*S*1*:sid*<*R*1*:sid R*1) are unnamed. Domains are inherited from the corresponding fields of *S*1 and *R*1.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

Figure 4.12 *S*1 ⋈ *S*1*: Sid*<*R*1*: Sid R*1

- **Equijoin**
- A common special case of the join operation *R* ⋈*S* is when the *join condition* consists solely of equalities (connected by ^) of the form *R: name1 = S: name2*, that is, equalities between two fields in *R* and *S*. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which *S:name*2 is dropped. The join operation with this refinement is called **equijoin**. The schema of the result of an equijoin contains the fields of *R* (with the same names and domains as in *R*) followed by the fields of *S* that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from *R* and *S*, they are unnamed in the result relation.
- We illustrate *S*1 ⋈*R:sid=S:sid R*1 in Figure 4.13. Notice that only one field called *sid* appears in the result.

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

Figure 4.13 *S*1 ⋈*R:sid=S:sid R*1

- **Natural Join**

- A further special case of the join operation *R ./ S* is an equijoin in which equalities are specified on *all* fields having the same name in *R* and *S*. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a *natural join*, and it has the nice property that the result is guaranteed not to have two fields with the same name.
- The equijoin expression *S*1 ⋈*R:sid=S:sid R*1 is actually a natural join and can simply be denoted as *S*1 ⋈ *R*1, since the only common field is *sid*. If the two relations have no attributes in common, *S*1 ⋈ *R*1 is simply the cross-product.

4.2.5 Division

The division operator is useful for expressing certain kinds of queries, for example:
''Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise. However, the division operator does not have the same importance as the other operators – it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator).

We discuss division through an example. Consider two relation instances *A* and *B* in which *A* has (exactly) two fields *x* and *y* and *B* has just one field *y*, with the same domain as in *A*. We define the *division* operation *A=B* as the set of all *x* values (in the form of unary tuples) such that for *every y* value in (a tuple of) *B*, there is a tuple *(z,y)* in *A*.

Another way to understand division is as follows. For each *x* value in (the first column of) *A*, consider the set of *y* values that appear in (the second field of) tuples of *A* with that *x* value. If this set contains (all *y* values in) *B*, the *x* value is in the result of *A=B*.

An analogy with integer division may also help to understand division. For integers *A* and *B*, *A=B* is the largest integer *Q* such that *Q* * *B* ≤*A*. For relation instances *A* and *B*, *A=B* is the largest relation instance *Q* such that *Q* × *B* ≤ *A*. Division is illustrated in Figure 4.14. It helps to think of *A* as a relation listing the parts supplied by suppliers, and of the *B* relations as listing parts. *A=Bi* computes suppliers who supply *all* parts listed in relation instance *Bi*.

Expressing *A=B* in terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all *x* values in *A* that are not *disqualified*. An *x* value is *disqualified* if by attaching a

A

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

y value from *B*, we obtain a tuple *(z,y)* that is not in *A*. We can compute disqualified tuples using the algebra expression

π2 x((*π2 x*(*A*) *x B*) − *A*)

Thus we can define *A=B* as

π2 x(*A*) − *π2 x*((*π2x*(*A*) *x B*) − *A*)

To understand the division operation in full generality, we have to consider the case when both *x* and *y* are replaced by a set of attributes. The generalization is straight for- ward and is left as an exercise for the reader. We will discuss two additional examples illustrating division (Queries Q9 and Q10) later in this section.

Expressing A/B Using Basic Operators

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially. Division is NOT implemented in SQL).
- *Idea*: For *SALES/PRODUCTS*, compute all products such that there exists at least one supplier not supplying it.
  - *x* value is *disqualified* if by attaching *y* value from *B*, we obtain an *xy* tuple that is not in *A*.

$$A = \pi_{sid}((\pi_{sid}(Sales) \times Products) - Sales)$$

The answer is $\pi_{sid}(Sales) - A$

Find names of sailors who've reserved boat #103

- Solution 1: $\pi_{sname}((\sigma_{bid=103} Reserves) \asymp Sailors)$
- Solution 2: $\rho(Temp1, \sigma_{bid=103} Reserves)$

  $\rho(Temp2, Temp1 \asymp Sailors)$

  $\pi_{sname}(Temp2)$
- Solution 3: $\pi_{sname}(\sigma_{bid=103}(Reserves \asymp Sailors))$

Find names of sailors who've reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}((\sigma_{color='red'} Boats) \asymp Reserves \asymp Sailors)$$

- A more efficient solution:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'} Boats) \asymp Res) \asymp Sailors)$$

A query optimizer can find this, given the first solution!

Find sailors who've reserved a red and a green boat

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'}, Boats) \times Reserves))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'}, Boats) \times Reserves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \times Sailors)$$

Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\ (Tempsids, (\pi_{sid,bid} Reserves) / (\pi_{bid} Boats))$$

$$\pi_{sname}\ (Tempsids \times Sailors)$$

- To find sailors who've reserved all 'Interlake' boats:

$$/\ \pi_{bid}(\sigma_{bname='Interlake'}\ Boats)$$

Q. Explain Relational Calculus In Detail?

- A **relational calculus** expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in **tuple calculus**) or over columns of the stored relations (in **domain calculus**).
- In a calculus expression, there is *no order of operations* to specify how to retrieve the query result—a calculus expression specifies only what information the result should contain.
- This is the main distinguishing feature between relational algebra and relational calculus.
- Relational calculus is considered to be a **nonprocedural** or **declarative** language.
- This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence relational algebra can be considered as a **procedural** way of stating a query.
- The tuple relational calculus is based on specifying a number of tuple variables.
- Each tuple variable usually ranges over a particular database relation, meaning that the variable may take as its value any individual tuple from that relation.
- A simple tuple relational calculus query is of the form
  {t | COND(t)}
- where t is a tuple variable and COND (t) is a conditional expression involving t.

- The result of such a query is the set of all tuples t that satisfy COND (t).
- Example: To find the first and last names of all employees whose salary is above $50,000, we can write the following tuple calculus expression:
- **{t.FNAME, t.LNAME | EMPLOYEE(t) AND t.SALARY>50000}**
- The condition EMPLOYEE(t) specifies that the **range relation** of tuple variable t is EMPLOYEE.
- The first and last name (PROJECTION $\mathbf{p}_{FNAME, LNAME}$) of each EMPLOYEE tuple t that satisfies the condition t.SALARY>50000 (SELECTION $\mathbf{s}_{SALARY >50000}$) will be retrieved.
- Two special symbols called quantifiers can appear in formulas; these are the universal quantifier (") and the existential quantifier ($).
- Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an (" t) or ($ t) clause; otherwise, it is free.
- If F is a formula, then so are ($ t)(F) and (" t)(F), where t is a tuple variable.
- The formula ($ t)(F) is true if the formula F evaluates to true for some (at least one) tuple assigned to free occurrences of t in F; otherwise ($ t)(F) is false.
- The formula (" t)(F) is true if the formula F evaluates to true for every tuple (in the universe) assigned to free occurrences of t in F; otherwise (" t)(F) is false.
- " is called the universal or “for all” quantifier because every tuple in “the universe of” tuples must make F true to make the quantified formula true.
- $ is called the existential or “there exists” quantifier because any tuple that exists in “the universe of” tuples may make F true to make the quantified formula true.
- Retrieve the name and address of all employees who work for the ‘Research’ department. The query can be expressed as :
- **{t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) and ($ d) (DEPARTMENT(d) and d.DNAME=‘Research’ and d.DNUMBER=t.DNO) }**
- The only *free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (|).
- In above query, t is the only free variable; it is then *bound successively* to each tuple.
- If a tuple *satisfies the conditions* specified in the query, the attributes FNAME, LNAME, and ADDRESS are retrieved for each such tuple.
- The conditions EMPLOYEE (t) and DEPARTMENT(d) specify the range relations for t and d.
- The condition d. DNAME = ‘Research’ is a selection condition and corresponds to a SELECT operation in the relational algebra, whereas the condition d. DNUMBER = t.DNO is a JOIN condition.
- Find the names of employees who work on *all* the projects controlled by department number 5. The query can be:
- **{e.LNAME, e.FNAME | EMPLOYEE(e) and ((" x)(not(PROJECT(x)) or not(x.DNUM=5)**
- **OR (($ w)(WORKS_ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO))))}**
- Exclude from the universal quantification all tuples that we are not interested in by making the condition true *for all such tuples*.
- The first tuples to exclude (by making them evaluate automatically to true) are those that are not in the relation R of interest.
- In query above, using the expression **not(PROJECT(x))** inside the universally quantified formula evaluates to true all tuples x that are not in the PROJECT relation.
- Then we exclude the tuples we are not interested in from R itself. The expression not(x.DNUM=5) evaluates to true all tuples x that are in the project relation but are not controlled by department 5.
- Finally, we specify a condition that must hold on all the remaining tuples in R.
- **(($ w)(WORKS_ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO)**

Quick Revision

- Relational algebra includes standard operations on sets such as union (*U*), intersection (∩), set-difference (−), and cross-product (×).
- Relations and fields can be renamed in relational algebra using the renaming operator (*ρ*).Another relational algebra operation that arises commonly in practice is the join (⋈) |with important special cases of equijoin and natural join.
- The division operation (=) is a convenient way to express that we only want tuples where all possible value combinations |as described in another relation| exist.
- Instead of describing a query by how to compute the output relation, a *relational calculus* query describes the tuples in the output relation. The language for specifying the output tuples is essentially a restricted subset of first-order predicate logic. In *tuple relational calculus*, variables take on tuple values and in *domain relational calculus*, variables take on field values, but the two versions of the calculus are very similar.
- All relational algebra queries can be expressed in relational calculus. If we restrict ourselves to safe queries on the calculus, the converse also holds. An important criterion for commercial query languages is that they should be *relationally complete* in the sense that they can express all relational algebra queries.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain selection and projection?	3
2	Q. write a note on set operations?	4
3	Q. Explain Relational Calculus In Detail?	8

CHAPTER 2: CREATING AND ALTERING TABLES

Topic covered:

- Conversion of ER to relations with and without constraints; CREATE statement with constraints like KEY, CHECK, DEFAULT, ALTER and DROP statement.

Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model. We begin by considering relationship sets without key and participation constraints, and we discuss how to handle such constraints in subsequent sections. To represent a relationship, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship. Thus, the attributes of the relation include:

The primary key attributes of each participating entity set, as foreign key fields.

The descriptive attributes of the relationship set.

The set of non descriptive attributes is a super key for the relation. If there are no key constraints (see Section 2.4.1), this set of attributes is a candidate key.

Consider the Works In2 relationship set shown in Figure 3.10. Each department has offices in several locations and we want to record the locations at which each employee works.

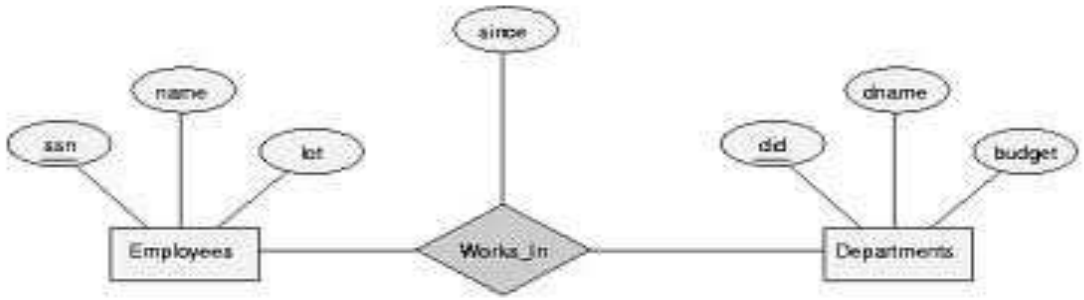


Figure 3.10 A Ternary Relationship Set

All the available information about the Works In2 table is captured by the following SQL definition:

```
CREATE TABLE Works In2 ( ssn CHAR(11), did INTEGER, address CHAR(20), since DATE, PRIMARY
                         KEY (ssn, did, address),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                         FOREIGN KEY (address) REFERENCES Locations,
                         FOREIGN KEY (did) REFERENCES Departments )
```

Note that the *address*, *did,* and *ssn* fields cannot take on *null* values. Because these fields are part of the primary key for Works In2, a NOT NULL constraint is implicit for each of these fields. This constraint ensures that these fields uniquely identify a department, an employee, and a location in each tuple of Works In. We can also specify that a particular action is desired when a referenced Employees, Departments or Locations tuple is deleted, as explained in the discussion of integrity constraints in Section 3.2. In this chapter we assume that the default action is appropriate except for situations in which the semantics of the ER diagram require some other action.

Finally, consider the Reports To relationship set shown in Figure 3.11. The role in- dicators *supervisor* and *subordinate* are used to create meaningful field names in the

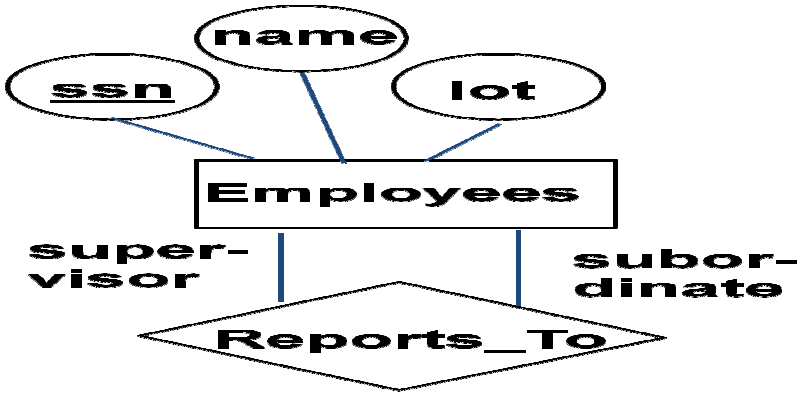


Figure 3.11 The Reports To Relationship Set

CREATE statement for the Reports To table:
CREATE TABLE Reports To (supervisor ssn CHAR(11), subordinate ssn CHAR(11),
PRIMARY KEY (supervisor ssn, subordinate ssn),
FOREIGN KEY (supervisor ssn) REFERENCES Employees(ssn),
FOREIGN KEY (subordinate ssn) REFERENCES Employees(ssn))

Observe that we need to explicitly name the referenced field of Employees because the field name differs from the name(s) of the referring field(s).

3.5.3 Translating Relationship Sets with Key Constraints

If a relationship set involves *n* entity sets and some *m* of them are linked via arrows in the ER diagram, the key for any one of these *m* entity sets constitutes a key for the relation to which the relationship set is mapped. Thus we have *m* candidate keys, and one of these should be designated as the primary key. The translation discussed in Section 2.3 from relationship sets to a relation can be used in the presence of key constraints, taking into account this point about keys.

Consider the relationship set Manages shown in Figure 3.12. The table corresponding

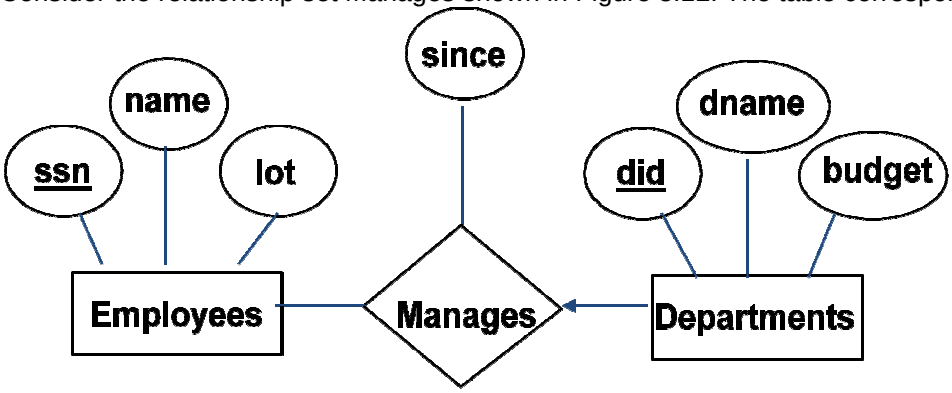


Figure 3.12 Key Constraint on Manages

to Manages has the attributes *ssn, did, since*. However, because each department has at most one manager, no two tuples can have the same *did* value but di_er on the *ssn* value. A consequence of this observation is that *did* is itself a key for Manages; indeed, the set *did, ssn* is not a key (because it is not minimal). The Manages relation can be de_ned using the following SQL statement:

CREATE TABLE Manages (ssn CHAR(11),
did INTEGER,
since DATE,
PRIMARY KEY (did),

FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (did) REFERENCES Departments)
A second approach to translating a relationship set with key constraints is often superior because it avoids creating a distinct table for the relationship set. The idea is to include the information about the relationship set in the table corresponding to the entity set with the key, taking advantage of the key constraint. In the Manages example, because a department has at most one manager, we can add the key fields of the Employees tuple denoting the manager and the *since* attribute to the Departments tuple. This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations. The only drawback to this approach is that space could be wasted if several departments have no managers. In this case the added fields would have to be filled with *null* values. The first translation (using a separate table for Manages) avoids this inefficiency, but some important queries require us to combine information from two relations, which can be a slow operation.
The following SQL statement, defining a Dept Mgr relation that captures the information in both Departments and Manages, illustrates the second approach to translating relationship sets with key constraints:

```
CREATE TABLE Dept Mgr ( did INTEGER,
                        dname CHAR(20),
                      budget REAL,
                      ssn CHAR(11),
                      since DATE,
                      PRIMARY KEY (did),
                      FOREIGN KEY (ssn) REFERENCES Employees )
```

Note that *ssn* can take on *null* values.
This idea can be extended to deal with relationship sets involving more than two entity sets. In general, if a relationship set involves *n* entity sets and some *m* of them are linked via arrows in the ER diagram, the relation corresponding to any one of the *m* sets can be augmented to capture the relationship.
We discuss the relative merits of the two translation approaches further after considering how to translate relationship sets with participation constraints into tables.

- Database constraints are restrictions on the contents of the database or on database operations

- Database constraints provide a way to guarantee that:
  - rows in a table have valid primary or unique key values
  - rows in a dependent table have valid foreign key values that reference rows in a parent table
  - individual column values are valid
- SQL/400 constraints cover four specific integrity rules:
  - primary key constraint (to enforce existence integrity)
  - unique constraint (to enforce candidate key integrity)
  - foreign key constraint (to enforce foreign key, or referential integrity)
  - check constraint (to restrict a column's values; a partial enforcement of domain integrity)
  - You specify one or more of these constraints when you use the Create Table statement to create a base table.
- You can also add or drop constraints with the Alter Table statement.
- UDB/400 enforces all four types of constraints when rows in the table are inserted or updated
- In the case of a foreign key constraint ➔ when rows in the parent table are updated or deleted
- Once a constraint for a table is defined
- The constraint is enforced for *all* database updates through *any* interface
- including SQL DML, HLL built-in I/O operations, Interactive SQL (ISQL) ad hoc updates, etc.
- Create Table Sale

- (OrderID Dec(7, 0) Not Null

- **Constraint** SaleOrderIdChk Check(OrderID > 0),

- SaleDate Date Not Null,
- ShipDate Date Default Null,
- SaleTot Dec(7, 2) Not Null,
- CrdAutNbr Int Default Null,
- CustID Dec(7, 0) Not Null,
- Primary Key(OrderID),
- **Constraint** SaleCrdAutNbrUK Unique (CrdAutNbr),
- **Constraint** SaleCustomerFK Foreign Key (CustID)
- References Customer (CustID)
- On Delete Cascade
- On Update Restrict,
- **Constraint** SaleShipDateChk Check(ShipDate Is Null Or ShipDate >= SaleDate))

- If you don't specify a constraint name, UDB/400 generates a name when the table is created.
  - The constraint name can later be used in the Alter Table statement to drop the constraint

Add/Remove Constraints

- After you create a table, you can use the Alter Table statement to
  - add or remove a primary key, unique, foreign key, or check constraint

- To drop a table's primary key constraint, just specify the Primary Key keywords:

```
Alter Table Sale
    Drop Primary Key
```

- To drop a unique, foreign key, or check constraint, you must specify the constraint name:

```
Alter Table Sale
    Drop Constraint SaleCustomerFK
```

- To add a new constraint, use the same constraint syntax as in a Create Table statement:

```
Alter Table Sale
    Add Constraint SaleSaleTotChk Check( SaleTot >= 0 )
```

- The ALTER TABLE statement is used to make changes to constraints in existing tables.

  - These changes can include adding or dropping constraints, enabling or disabling constraints, and adding a NOT NULL constraint to a column.
  - The guidelines for making changes to constraints are:
    - You can add, drop, enable, or disable a constraint, but you cannot modify its structure.
    - You can add a NOT NULL constraint to an existing column by using the MODIFY clause of the ALTER TABLE statement. MODIFY is used because NOT NULL is a column-level change.
    - You can define a NOT NULL constraint only if the table is empty or if the column has a value for every row.
- **The ALTER TABLE statement requires:**
  - name of the table
  - name of the constraint
  - type of constraint
  - name of the column affected by the constraint
- In the code example shown below, using the DJs on Demand database, the primary-key constraint could have been added after the D_CLIENTS table was originally created. In this case, the primary-key constraint is being added to the D_CLIENTS table.

ALTER TABLE d_clients

- To add a constraint to an existing table, use the following SQL syntax:

ALTER TABLE table_name
ADD [CONSTRAINT constraint_name] type of constraint(column_name);

  - If the constraint is a FOREIGN KEY constraint, the REFERENCES keyword must be included in the statement:

ADD CONSTRAINT constraint_name FOREIGN
KEY(column_name) REFERENCES tablename(column_name);
ALTER TABLE table_name

DROP CONSTRAINT name [CASCADE];

- To drop a constraint,
  - you can identify the constraint name from the USER_CONSTRAINTS and USER_CONS_COLUMNS in the data dictionary.
  - The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.
  - Note that
    - when you drop an integrity constraint, that constraint is no longer enforced by the Oracle Server and is no longer available in the data dictionary.
- By default,
  - whenever an integrity constraint is defined in a CREATE or ALTER TABLE statement,
  - the constraint is automatically enabled (enforced) by Oracle unless it is specifically created in a disabled state using the DISABLE clause.
- You can disable a constraint without dropping it or re-creating it by using the ALTER TABLE option DISABLE.
  - DISABLE allows incoming data, regardless of whether it conforms to the constraint.
  - This function allows data to be added to a child table, without having corresponding values in the parent table.

CHAPTER 3: HANDLING DATA USING SQL

Topic Covered:

- selecting data using SELECT statement, FROM clause, WHERE clause, HAVING clause,
- ORDER BY, GROUP BY,DISTINCT and ALL predicates, Adding data with INSERT statement, changing data with UPDATE statement, removing data with DELETE statement

Q. Write a short note on SELECT statement?

- **The SELECT Statement**

- The SELECT statement retrieves data from a database and returns it to you in the form of query results. You have already seen many examples of the SELECT statement in the quick tour presented in Chapter 2. Here are several more sample queries that retrieve information about sales offices:
- *List the sales offices with their targets and actual sales.*

```
SELECT CITY, TARGET, SALES
FROM OFFICES
CITY TARGET SALES
------------- ---------- -----------
Denver $300,000.00 $186,042.00
New York $575,000.00 $692,637.00
Chicago $800,000.00 $735,042.00
Atlanta $350,000.00 $367,911.00
Los Angeles $725,000.00 $835,915.00
```

List the Eastern region sales offices with their targets and sales.

```
SELECT CITY, TARGET, SALES
FROM OFFICES
WHERE REGION = 'Eastern'
CITY TARGET SALES
------------- ----------- ------------
New York $575,000.00 $692,637.00
Chicago $800,000.00 $735,042.00
Atlanta $350,000.00 $367,911.00
```

List Eastern region sales offices whose sales exceed their targets, sorted in alphabetical order by city.

```
SELECT CITY, TARGET, SALES
FROM OFFICES
WHERE REGION = 'Eastern'
AND SALES > TARGET
ORDER BY CITY
CITY TARGET SALES
------------- ----------- ----------
Atlanta $350,000.00 $367,911.00
New York $575,000.00 $692,637.00
```

What are the average target and sales for Eastern region offices?

```
SELECT AVG(TARGET), AVG(SALES)
FROM OFFICES
WHERE REGION = 'Eastern'
AVG(TARGET) AVG(SALES)
------------- ----------
$575,000.00 $598,530.00
```

For simple queries, the English language request and the SQL SELECT statement are very similar. When the requests become more complex, more features of the SELECT statement must be used to specify the query precisely.

Figure 6-1 shows the full form of the SELECT statement, which consists of six clauses. The SELECT and FROM clauses of the statement are required. The remaining four clauses are optional. You include them in a SELECT statement only when you want to use the functions they provide. The following list summarizes the function of each clause:

- The SELECT clause lists the data items to be retrieved by the SELECT statement. The items may be columns from the database, or columns to be calculated by SQL as it performs the query. The SELECT clause is described in later sections of this chapter.
- **The SELECT Clause**
  The SELECT clause that begins each SELECT statement specifies the data items to be retrieved by the query. The items are usually specified by a *select list,* a list of *select items* separated by commas. Each select item in the list generates a single column of query results, in left-to-right order. A select item can be: a *column name*, identifying a column from the table(s) named in the FROM clause. When a column name appears as a select item, SQL simply takes the value of that column from each row of the database table and places it in the corresponding row of query results. a *constant*, specifying that the same constant value is to appear in every row of the query results. a *SQL expression*, indicating that SQL must calculate the value to be placed into the query results, in the style specified by the expression.
  Each type of select item is described later in this chapter.
- **Selecting All Columns (SELECT *)**
  Sometimes it's convenient to display the contents of all the columns of a table. This can be particularly useful when you first encounter a new database and want to get a quick Understanding of its structure and the data it contains. As a convenience, SQL lets you Use an asterisk (*) in place of the select list as an abbreviation for "all columns":
  Show me all the data in the OFFICES table.
  SELECT * FROM OFFICES
  OFFICE CITY REGION MGR TARGET SALES
  ------------------- -------- --- ----------- -----------
  22 Denver Western 108 $300,000.00 $186,042.00
  11 New York Eastern 106 $575,000.00 $692,637.00
  12 Chicago Eastern 104 $800,000.00 $735,042.00
  13 Atlanta Eastern 105 $350,000.00 $367,911.00
  21 Los Angeles Western 108 $725,000.00 $835,915.00
  The query results contain all six columns of the OFFICES table, in the same left-to-right order as in the table itself.
  The ANSI/ISO SQL standard specifies that a SELECT statement can have either an allcolumn selection or a select list, but not both, as shown in Figure 6-1. However, many SQL implementations treat the asterisk (*) as just another element of the select list. Thus the query:
  SELECT *, (SALES - TARGET)
  FROM OFFICES
  is legal in most commercial SQL dialects (for example in DB2, Oracle, and SQL Server), but it is not permitted by the ANSI/ISO standard.
- The all-columns selection is most appropriate when you are using interactive SQL casually. It should be avoided in programmatic SQL, because changes in the database structure can cause a program to fail. For example, suppose the OFFICES table were dropped from the database and then re-created with its columns rearranged and a new seventh column added. SQL automatically takes care of the database-related details of such changes, but it cannot modify your application program for you. If your program expects a SELECT * FROM OFFICES query to return six columns of query results with certain data types, it will almost certainly stop working when the columns are rearranged and a new one is added.
- These difficulties can be avoided if you write the program to request the columns it needs by name. For example, the following query produces the same results as SELECT * FROM OFFICES. It is also immune to changes in the database structure, as long as the named columns continue to exist in the OFFICES table: SELECT OFFICE, CITY, REGION, MGR, TARGET, SALES FROM OFFICES

Q. Write a short note on FROM Clause?

- **The FROM Clause**
  - The FROM clause consists of the keyword FROM, followed by a list of table specifications separated by commas. Each table specification identifies a table containing data to be retrieved by the query. These tables are called the *source tables* of the query (and of the SELECT statement) because they

are the source of all of the data in the query results. All of the queries in this chapter have a single source table, and every FROM clause contains a single table name.

- The FROM clause lists the tables that contain the data to be retrieved by the query. Queries that draw their data from a single table are described in this chapter. More complex queries that combine data from two or more tables are discussed in Chapter 7.
- **Simple Queries**

  The simplest SQL queries request columns of data from a single table in the database.
  For example, this query requests three columns from the OFFICES table:
  List the location, region, and sales of each sales office.

```
SELECT CITY, REGION, SALES
FROM OFFICES
CITY REGION SALES
-------------- ------- ------------
Denver Western $186,042.00
New York Eastern $692,637.00
Chicago Eastern $735,042.00
Atlanta Eastern $367,911.00
Los Angeles Western $835,915.00
```

  The SELECT statement for simple queries like this one includes only the two required clauses. The SELECT clause names the requested columns; the FROM clause names the table that contains them.
  Conceptually, SQL processes the query by going through the table named in the FROM clause, one row at a time, as shown in Figure 6-3. For each row, SQL takes the values of the columns requested in the select list and produces a single row of query results. The query results thus contain one row of data for each row in the table.

Q. Write a short note on WHERE clause?

- The WHERE clause tells SQL to include only certain rows of data in the query results. A *search condition* is used to specify the desired rows. The basic uses of the WHERE clause are described later in this chapter. Those that involve sub queries are discussed in Chapter 9.
- A WHERE clause in SQL specifies that a SQL Data Manipulation Language (DML) statement should only affect rows that meet specified criteria. The criteria are expressed in the form of predicates. WHERE clauses are not mandatory clauses of SQL DML statements, but should be used to limit the number of rows affected by a SQL DML statement or returned by a query.
  WHERE is an SQL reserved word.
  The WHERE clause is used in conjunction with SQL DML statements, and takes the following general form:

```
SQL-DML-Statement
FROM TABLE_NAME
WHERE predicate
```

  all rows for which the predicate in the WHERE clause is True are affected (or returned) by the SQL DML statement or query. Rows for which the predicate evaluates to False or Unknown (NULL) are unaffected by the DML statement or query.
  The following query returns only those rows from table *mytable* where the value in column *mycol* is greater than 100.

```
SELECT *  FROM   mytable
WHERE  mycol > 100
```

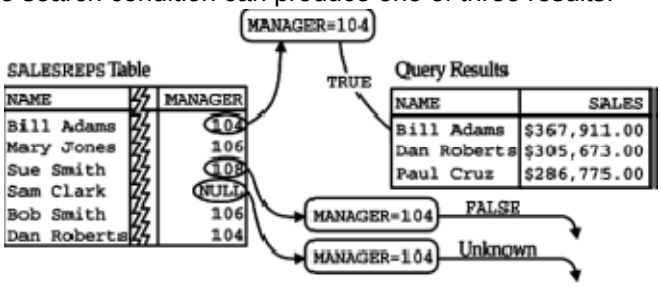
- **Row Selection (WHERE Clause)**

  SQL queries that retrieve all rows of a table are useful for database browsing and reports, but for little else. Usually you'll want to select only some of the rows in a table and include only these rows in the query results. The WHERE clause is used to specify the rows you want to retrieve. Here are some examples of simple queries that use the WHERE clause:
  Show me the offices where sales exceed target.

```
SELECT CITY, SALES, TARGET
FROM OFFICES
WHERE SALES > TARGET
```

CITY SALES TARGET
---------- ---------- ----------
New York $692,637.00 $575,000.00
Atlanta $367,911.00 $350,000.00
Los Angeles $835,915.00 $725,000.00
Show me the name, sales, and quota of employee number 105.
SELECT NAME, SALES, QUOTA
FROM SALESREPS
WHERE EMPL_NUM = 105
NAME SALES QUOTA
---------- ---------- ----------
Bill Adams $367,911.00 $350,000.00
Show me the employees managed by Bob Smith (employee 104).
SELECT NAME, SALES
FROM SALESREPS
WHERE MANAGER = 104
NAME SALES
---------- ----------
Bill Adams $367,911.00
Dan Roberts $305,673.00
Paul Cruz $286,775.00

- The WHERE clause consists of the keyword WHERE followed by a search condition that specifies the rows to be retrieved. In the previous query, for example, the search condition is MANAGER = 104. Figure 6-5 shows how the WHERE clause works.
- Conceptually, SQL goes through each row of the SALESREPS table, one-by-one, and applies the search condition to the row. When a column name appears in the search condition (such as the MANAGER column in this example), SQL uses the value of the column in the current row. For each row, the search condition can produce one of three results:



- **Figure 6-5:** Row selection with the WHERE clause

Q. Write a short note on HAVING Clause?

- **Having (SQL)**
- A **HAVING** clause in SQL specifies that an SQL SELECT statement should only return rows where aggregate values meet the specified conditions. It was added to the SQL language because the WHERE keyword could not be used with aggregate functions.

- **Examples:**

- **To return a list of department IDs whose total sales exceeded $1000 on the date of January 1, 2000, along with the sum of their sales on that date:**

  SELECT DeptID, SUM(SaleAmount)
  FROM Sales
  WHERE SaleDate = '01-Jan-2000'
  GROUP BY DeptID
  HAVING SUM(SaleAmount) > 1000

  Referring to the sample tables in the Join (SQL) example, the following query will return the list of departments who have more than 1 employee:

```
SELECT DepartmentName, COUNT(*)
FROM employee,department
WHERE employee.DepartmentID = department.DepartmentID
GROUP BY DepartmentName
HAVING COUNT(*)>1;
```

Q. Write a short note on ORDER By and GROUP BY?

- An **ORDER BY** clause in SQL specifies that a SQL SELECT statement returns a result set with the rows being sorted by the values of one or more columns. The sort criteria do not have to be included in the result set. The sort criteria can be expressions, including – but not limited to – column names, user-defined functions, arithmetic operations, or CASE expressions. The expressions are evaluated and the results are used for the sorting, i.e. the values stored in the column or the results of the function call.
- ORDER BY is the *only* way to sort the rows in the result set. Without this clause, the relational database system may return the rows in any order. If an ordering is required, the ORDER BY must be provided in the SELECT statement sent by the application. Although some database systems allow the specification of an ORDER BY clause in subselects or view definitions, the presence there has no effect. A view is a logical relational table, and the relational model mandates that a table is a set of rows, implying no sort order whatsoever. The only exception are constructs like ORDER BY ORDER OF ... (not standardized in SQL:2003) which allow the propagation of sort criteria through nested subselects.
- The SQL standard's core functionality does not explicitly define a default sort order for Nulls. With the SQL:2003 extension T611, "Elementary OLAP operations", nulls can be sorted before or after all data values by using the NULLS FIRST or NULLS LAST clauses of the ORDER BY list, respectively. Not all DBMS vendors implement this functionality, however. Vendors who do not implement this functionality may specify different treatments for Null sorting in the DBMS.[1]
- Structure **ORDER BY ... DESC** will order in descending order, otherwise ascending order is used. (The latter may be specified explicitly using **ASC**.)

- **Examples**

```
SELECT * FROM Employees
ORDER BY LastName, FirstName
```

This sorts by the LastName field, then by the FirstName field if LastName matches.

➢ **GROUP BY**

- The GROUP BY clause is used to project rows having common values into a smaller set of rows. GROUP BY is often used in conjunction with SQL aggregation functions or to eliminate duplicate rows from a result set. The WHERE clause is applied before the GROUP BY clause.
- The example below demonstrates a query of multiple tables, grouping, and aggregation, by returning a list of books and the number of authors associated with each book.
- SELECT Book.title,
- COUNT(*) AS Authors

```
FROM  Book JOIN Book_author
ON Book.isbn = Book_author.isbn
GROUP BY Book.title;
```

Example output might resemble the following:

```
Title                Authors
-------------------- -------
SQL Examples and Guide     4
The Joy of SQL             1
An Introduction to SQL     2
Pitfalls of SQL            1
```

- Under the precondition that *isbn* is the only common column name of the two tables and that a column named *title* only exists in the *Books* table, the above query could be rewritten in the following form:

  SELECT title,

  COUNT(*) AS Authors

  FROM Book NATURAL JOIN Book_author

  GROUP BY title;

- However, many vendors either do not support this approach, or require certain column naming conventions in order for natural joins to work effectively.
- SQL includes operators and functions for calculating values on stored values. SQL allows the use of expressions in the *select list* to project data, as in the following example which returns a list of books that cost more than 100.00 with an additional *sales_tax* column containing a sales tax figure calculated at 6% of the *price*.

Q. Write a short note on DISTINCT?

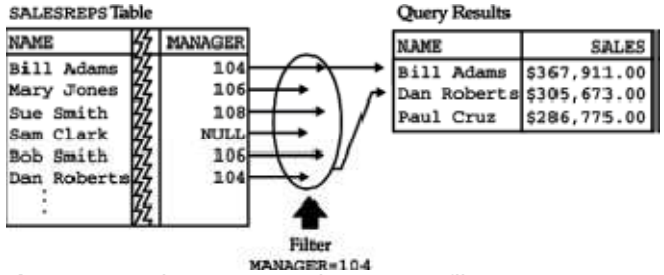
- **Duplicate Rows (DISTINCT)**
- If a query includes the primary key of a table in its select list, then every row of query results will be unique (because the primary key has a different value in each row). If the primary key is not included in the query results, duplicate rows can occur. For example, suppose you made this request:
  List the employee numbers of all sales office managers.
  SELECT MGR
  FROM OFFICES
  MGR

  108
  106
  104
  105
  108
- The query results have five rows (one for each office), but two of them are exact duplicates of one another. Why? Because Larry Fitch manages both the Los Angeles and Denver offices, and his employee number (108) appears in both rows of the OFFICES table. These query results are probably not exactly what you had in mind. If there are four different managers, you might have expected only four employee numbers in the query results.
- You can eliminate duplicate rows of query results by inserting the keyword DISTINCT in the SELECT statement just before the select list. Here is a version of the previous query that produces the results you want:
  List the employee numbers of all sales office managers.
  SELECT DISTINCT MGR
  FROM OFFICES
  MGR

  104
  105
  106
  108
- Conceptually, SQL carries out this query by first generating a full set of query results (five rows) and then eliminating rows that are exact duplicates of one another to form the final query results. The DISTINCT keyword can be specified regardless of the contents of the

SELECT list (with certain restrictions for summary queries, as described in Chapter 8). If the DISTINCT keyword is omitted, SQL does not eliminate duplicate rows. You can also specify the keyword ALL to explicitly indicate that duplicate rows are to be retained, but it is unnecessary since this is the default behavior.

- If the search condition is TRUE, the row is included in the query results. For example, the row for Bill Adams has the correct MANAGER value and is included.
- If the search condition is FALSE, the row is excluded from the query results. For example, the row for Sue Smith has the wrong MANAGER value and is excluded.
- If the search condition has a NULL (unknown) value, the row is excluded from the query results. For example, the row for Sam Clark has a NULL value for the MANAGER column and is excluded.



- **Figure 6-6:** The WHERE clause as a filter
- Figure 6-6 shows another way to think about the role of the search condition in the
- WHERE clause. Basically, the search condition acts as a filter for rows of the table. Rows
- that satisfy the search condition pass through the filter and become part of the query
- results. Rows that do not satisfy the search condition are trapped by the filter and
- excluded from the query results.

Q. Write a short note on INSERT statement?

- A *single-row* INSERT statement adds a single new row of data to a table. It is commonly used in daily applications—for example, data entry programs.
- A *multi-row* INSERT statement extracts rows of data from another part of the databaseand adds them to a table. It is commonly used in end-of-month or end-of-year processing when "old" rows of a table are moved to an inactive table.
- A *bulk load* utility adds data to a table from a file that is outside of the database. It is commonly used to initially load the database or to incorporate data downloaded from another computer system or collected from many sites.
- **<u>The Single-Row INSERT Statement</u>**
- The single-row INSERT statement, shown in Figure 10-1, adds a new row to a table. The INTO clause specifies the table that receives the new row (the *target* table), and the VALUES clause specifies the data values that the new row will contain. The column list indicates which data value goes into which column of the new row

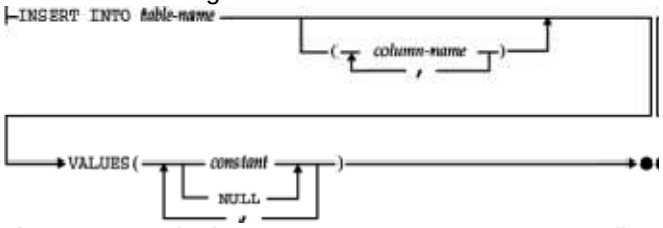


Figure 10-1: Single-row INSERT statement syntax diagram

- Suppose you just hired a new salesperson, Henry Jacobsen, with the following personal data:
  Name: Henry Jacobsen
  Age:
  3 6
  Employee Number:
  1 11
  Title:

Sales Manager
Office:
Atlanta (office number 13)
Hire Date:
J uly 25, 1990
Quota:
N ot yet assigned
Year-to-Date Sales:
$ 0.00

- Here is the INSERT statement that adds Mr. Jacobsen to the sample database:
  Add Henry Jacobsen as a new salesperson.
  INSERT INTO SALESREPS (NAME, AGE, EMPL_NUM, SALES, TITLE, HIRE_DATE, REP_OFFICE)
  VALUES ('Henry Jacobsen', 36, 111, 0.00, 'Sales Mgr',
  '25-JUL-90', 13)
  1 row inserted.
- Figure 10-2 graphically illustrates how SQL carries out this INSERT statement.
- Conceptually, the INSERT statement builds a single row of data that matches the column structure of the table, fills it with the data from the VALUES clause, and then adds the new row to the table. The rows of a table are unordered, so there is no notion of inserting the row "at the top" or "at the bottom" or "between two rows" of the table. After the INSERT statement, the new row is simply a part of the table. A subsequent query against the SALESREPS table will include the new row, but it may appear anywhere among the rows of query results.

- **Figure 10-2:** Inserting a single row
- Suppose that Mr. Jacobsen now receives his first order, from InterCorp, a new customer who is assigned customer number 2126. The order is for 20 ACI-41004 Widgets, for a total price of $2,340, and has been assigned order number 113069. Here are the INSERT statements that add the new customer and the order to the database:
  Insert a new customer and order for Mr. Jacobsen.
  INSERT INTO CUSTOMERS (COMPANY, CUST_NUM, CREDIT_LIMIT, CUST_REP)
  VALUES ('InterCorp', 2126, 15000.00, 111)
  1 row inserted.
  INSERT INTO ORDERS (AMOUNT, MFR, PRODUCT, QTY, ORDER_DATE, ORDER_NUM, CUST, REP)
  VALUES (2340.00, 'ACI', '41004', 20, CURRENT DATE, 113069, 2126, 111)
  1 row inserted.
  As this example shows, the INSERT statement can become lengthy if there are many columns of data, but its format is still very straightforward. The second INSERT statement uses the system constant CURRENT DATE in its VALUES clause, causing the current date to be inserted as the order date. This system constant is specified in the SQL2 standard and is supported by many of the popular SQL products. Other brands of DBMS provide other system constants or built-in functions to obtain the current date and time.
  You can use the INSERT statement with interactive SQL to add rows to a table that

grows very rarely, such as the OFFICES table. In practice, however, data about a new customer, order, or salesperson is almost always added to a database through a formsoriented data entry program. When the data entry is complete, the application program inserts the new row of data using programmatic SQL. Regardless of whether interactive or programmatic SQL is used, however, the INSERT statement is the same.

- The table name specified in the INSERT statement is normally an unqualified table name, specifying a table that you own. To insert data into a table owned by another user, you can specify a qualified table name. Of course you must also have permission to insert data into the table, or the INSERT statement will fail. The SQL security scheme and permissions are described in Chapter 15.
- The purpose of the column list in the INSERT statement is to match the data values in the VALUES clause with the columns that are to receive them. The list of values and the list of columns must both contain the same number of items, and the data type of each value must be compatible with the data type of the corresponding column, or an error will occur.
- The ANSI/ISO standard mandates unqualified column names in the column list, but many implementations allow qualified names. Of course, there can be no ambiguity in the column names anyway, because they must all reference columns of the target table.

The Multi-Row INSERT Statement

The second form of the INSERT statement, shown in Figure 10-3, adds multiple rows of data to its target table. In this form of the INSERT statement, the data values for the new rows are not explicitly specified within the statement text. Instead, the source of new rows is a database query, specified in the statement.

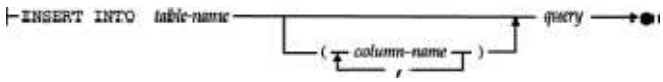


Figure 10-3: Multi-row INSERT statement syntax diagram

Adding rows whose values come from within the database itself may seem strange at first, but it's very useful in some special situations. For example, suppose that you want to copy the order number, date, and amount of all orders placed before January 1, 1990, from the ORDERS table into another table, called OLDORDERS. The multi-row INSERT statement provides a compact, efficient way to copy the data: *Copy old orders into the OLDORDERS table.*

```
INSERT INTO OLDORDERS (ORDER_NUM, ORDER_DATE, AMOUNT)
SELECT ORDER_NUM, ORDER_DATE, AMOUNT
FROM ORDERS
WHERE ORDER_DATE < '01-JAN-90'
9 rows inserted.
```

This INSERT statement looks complicated, but it's really very simple. The statement identifies the table to receive the new rows (OLDORDERS) and the columns to receive the data, just like the single-row INSERT statement. The remainder of the statement is a query that retrieves data from the ORDERS table. Figure 10-4 graphically illustrates the operation of this INSERT statement. Conceptually, SQL first performs the query against the ORDERS table and then inserts the query results, row by row, into the OLDORDERS table.

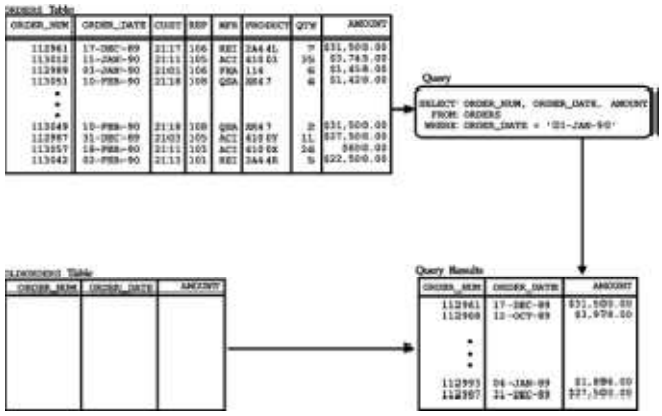


Figure 10-4: Inserting multiple rows

Here's another situation where you could use the multi-row INSERT statement. Suppose you want to analyze customer-buying patterns by looking at which customers and salespeople are responsible for big orders—those over $15,000. The queries that you will be running will combine data from the CUSTOMERS, SALESREPS, and ORDERS tables.

These three-table queries will execute fairly quickly on the small sample database, but in a real corporate database with many thousands of rows, they would take a long time.

Rather than running many long, three-table queries, you could create a new table named BIGORDERS to contain the required data, defined as follows:

Column Information

AMOUNT
Order amount (from ORDERS)
COMPANY
Customer name (from CUSTOMERS)
NAME
Salesperson name (from SALESREPS)
PERF
Amount over/under quota (calculated from SALESREPS)
MFR
Manufacturer id (from ORDERS)
PRODUCT
Product id (from ORDERS)
QTY
Quantity ordered (from ORDERS)

Once you have created the BIGORDERS table, this multi-row INSERT statement can be used to populate it:

Load data into the BIGORDERS table for analysis.

```
INSERT INTO BIGORDERS (AMOUNT, COMPANY, NAME, PERF, PRODUCT, MFR,
QTY)
SELECT AMOUNT, COMPANY, NAME, (SALES - QUOTA), PRODUCT, MFR,
QTY
FROM ORDERS, CUSTOMERS, SALESREPS
WHERE CUST = CUST_NUM
AND REP = EMPL_NUM
AND AMOUNT > 15000.00
```

Q. Write a short note on UPDATE statement?

- **The UPDATE Statement**
- The UPDATE statement, shown in Figure 10-6, modifies the values of one or more columns in selected rows of a single table. The target table to be updated is named in the statement, and you must have the required permission to update the table as well as each of the individual columns that will be modified. The WHERE clause selects the rows of the table to be modified. The SET clause specifies which columns are to be updated and calculates the new values for them.

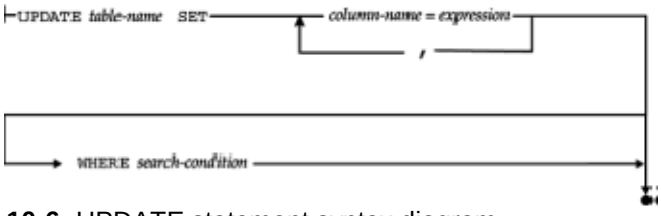


Figure 10-6: UPDATE statement syntax diagram

- Here is a simple UPDATE statement that changes the credit limit and salesperson for a customer:
- *Raise the credit limit for Acme Manufacturing to $60,000 and reassign them to Mary Jones (employee number 109).*

```
UPDATE CUSTOMERS
SET CREDIT_LIMIT = 60000.00, CUST_REP = 109
WHERE COMPANY = 'Acme Mfg.'
1 row updated.
```

- In this example, the WHERE clause identifies a single row of the CUSTOMERS table, and the SET clause assigns new values to two of the columns in that row. The WHERE clause is exactly the same one you would use in a DELETE or SELECT statement to identify the row. In fact, the search conditions that can appear in the WHERE clause of an UPDATE statement are exactly the same as those available in the SELECT and DELETE statements.
- Like the DELETE statement, the UPDATE statement can update several rows at once with the proper search condition, as in this example:
- *Transfer all salespeople from the Chicago office (number 12) to the New York office (number 11), and lower their quotas by 10 percent.*

```
UPDATE SALESREPS
SET REP_OFFICE = 11, QUOTA = .9 * QUOTA
WHERE REP_OFFICE = 12
3 rows updated.
```

- In this case, the WHERE clause selects several rows of the SALESREPS table, and the value of the REP_OFFICE and QUOTA columns are modified in all of them. Conceptually, SQL processes the UPDATE statement by going through the SALESREPS table row by row, updating those rows for which the search condition yields a TRUE result and skipping over those for which the search condition yields a FALSE or NULL result. Because it searches the table, this form of the UPDATE statement is sometimes called a *searched* UPDATE statement. This term distinguishes it from a different form of the UPDATE statement, called a positioned UPDATE statement, which always updates a single row.
- The positioned UPDATE statement applies only to programmatic SQL and is described in Chapter 17.
- Here are some additional examples of searched UPDATE statements:
  Reassign all customers served by employee number 105, 106, or 107 to employee number 102.

```
UPDATE CUSTOMERS
SET CUST_REP = 102
WHERE CUST_REP IN (105, 106, 107)
5 rows updated.
```

  Assign a quota of $100,000 to any salesperson who currently has no quota.

```
UPDATE SALESREPS
SET QUOTA = 100000.00
WHERE QUOTA IS NULL
1 row updated.
```

- The SET clause in the UPDATE statement is a list of assignments separated by commas. Each assignment identifies a target column to be updated and specifies how to calculate the new value for the target column. Each target column should appear only once in the list; there should not be two assignments for the same target column. The ANSI/ISO specification mandates unqualified names for the target columns, but some SQL implementations allow qualified column names. There can be no ambiguity in the column names anyway, because they must refer to columns of the target table.
- The expression in each assignment can be any valid SQL expression that yields a value of the appropriate data type for the target column. The expression must be computable based on the values of the row currently being updated in the target table. In most DBMS implementations, the expression may not include any column functions or subqueries.
- If an expression in the assignment list references one of the columns of the target table, the value used to calculate the expression is the value of that column in the current row *before* any updates are applied. The same is true of column references that occur in the WHERE clause. For example, consider this (somewhat contrived) UPDATE statement:

```
UPDATE OFFICES
```

SET QUOTA = 400000.00, SALES = QUOTA
WHERE QUOTA < 400000.00
Before the update, Bill Adams had a QUOTA value of $350,000 and a SALES value of $367,911. After the update, his row has a SALES value of $350,000, *not* $400,000. The order of the assignments in the SET clause is thus immaterial; the assignments can be specified in any order.

- **Updating All Rows**
- The WHERE clause in the UPDATE statement is optional. If the WHERE clause is omitted, then *all* rows of the target table are updated, as in this example:
  Raise all quotas by 5 percent.
  UPDATE SALESREPS
  SET QUOTA = 1.05 * QUOTA
  10 rows updated.
  Unlike the DELETE statement, in which the WHERE clause is almost never omitted, the UPDATE statement without a WHERE clause performs a useful function. It basically performs a bulk update of the entire table, as demonstrated in the preceding example.
- **UPDATE with Subquery ***
  As with the DELETE statement, subqueries can play an important role in the UPDATE statement because they let you select rows to update based on information contained in other tables. Here are several examples of UPDATE statements that use subqueries:
  Raise by $5,000 the credit limit of any customer who has placed an order for more than $25,000.
  UPDATE CUSTOMERS
  SET CREDIT_LIMIT = CREDIT_LIMIT + 5000.00
  WHERE CUST_NUM IN (SELECT DISTINCT CUST
  FROM ORDERS
  WHERE AMOUNT > 25000.00)
  4 rows updated.

Reassign all customers served by salespeople whose sales are less than 80 percent of their quota.

```
UPDATE CUSTOMERS
SET CUST_REP = 105
WHERE CUST_REP IN (SELECT EMPL_NUM
FROM SALESREPS
WHERE SALES < (.8 * QUOTA))
2 rows updated.
```

Have all salespeople who serve over three customers report directly to Sam Clark (employee number 106).

```
UPDATE SALESREPS
SET MANAGER = 106
WHERE 3 < (EECT COUNT(*)
FROM CUSTOMERS
WHERE CUST_REP = EMPL_NUM)
1 row updated.
```

- As in the DELETE statement, subqueries in the WHERE clause of the UPDATE statement can be nested to any level and can contain outer references to the target table of the UPDATE statement. The column EMPL_NUM in the subquery of the preceding example is such an outer reference; it refers to the EMPL_NUM column in the row of the SALESREPS table currently being checked by the UPDATE statement. The subquery in this example is a correlated subquery, as described in Chapter 9.
- Outer references will often be found in subqueries of an UPDATE statement, because they implement the "join" between the table(s) in the subquery and the target table of the UPDATE statement. The same SQL1 restriction applies as for the DELETE statement: the target table cannot appear in the FROM clause of any subquery at any level of nesting. This prevents the subqueries from referencing the target table (some of whose rows may have already been updated). Any references to the target table in the subqueries are thus outer references to the row of the target table currently being tested by the UPDATE statement's WHERE clause. The SQL2 standard again removed this restriction and

specifies that a reference to the target table in a subquery is evaluated as if none of the target table had been updated.

Q. Write a short note on DELETE Statement?

- **The DELETE Statement**
- The DELETE statement, shown in Figure 10-5, removes selected rows of data from a single table. The FROM clause specifies the target table containing the rows. The WHERE clause specifies which rows of the table are to be deleted.

Figure 10-5: DELETE statement syntax diagram

- Suppose that Henry Jacobsen, the new salesperson hired earlier in this chapter, has just decided to leave the company. Here is the DELETE statement that removes his row from the SALESREPS table:
  Remove Henry Jacobsen from the database.
  DELETE FROM SALESREPS
  WHERE NAME = 'Henry Jacobsen'
  1 row deleted.
- The WHERE clause in this example identifies a single row of the SALESREPS table, which SQL removes from the table. The WHERE clause should have a familiar appearance—it's exactly the same WHERE clause that you would specify in a SELECT statement to *retrieve* the same row from the table. The search conditions that can be specified in the WHERE clause of the DELETE statement are the same ones available in the WHERE clause of the SELECT statement, as described in Chapters 6 and 9.
  Recall that search conditions in the WHERE clause of a SELECT statement can specify a single row or an entire set of rows, depending on the specific search condition. The same is true of the WHERE clause in a DELETE statement. Suppose, for example, that Mr. Jacobsen's customer, InterCorp (customer number 2126), has called to cancel all of their orders. Here is the delete statement that removes the orders from the ORDERS table:
  Remove all orders for InterCorp (customer number 2126).
  DELETE FROM ORDERS
  WHERE CUST = 2126
  2 rows deleted.
- In this case, the WHERE clause selects several rows of the ORDERS table, and SQL removes all of the selected rows from the table. Conceptually, SQL applies the WHERE clause to each row of the ORDERS table, deleting those where the search condition yields a TRUE result and retaining those where the search condition yields a FALSE or NULL result. Because this type of DELETE statement searches through a table for the rows to be deleted, it is sometimes called a *searched* DELETE statement. This term is used to contrast it with another form of the DELETE statement, called the positioned DELETE statement, which always deletes a single row. The positioned DELETE statement applies only to programmatic SQL and is described in Chapter 17.
  Here are some additional examples of searched DELETE statements:
  Delete all orders placed before November 15, 1989.
  DELETE FROM ORDERS
  WHERE ORDER_DATE < '15-NOV-89'
  5 rows deleted.
  Delete all rows for customers served by Bill Adams, Mary Jones, or Dan Roberts (employee numbers 105, 109, and 101).
  DELETE FROM CUSTOMERS
  WHERE CUST_REP IN (105, 109, 101)
  7 rows deleted.
  Delete all salespeople hired before July 1988 who have not yet been assigned a quota.
  DELETE FROM SALESREPS
  WHERE HIRE_DATE < '01-JUL-88'

AND QUOTA IS NULL
0 rows deleted.

- **Deleting All Rows**
- The WHERE clause in a DELETE statement is optional, but it is almost always present. If the WHERE clause is omitted from a DELETE statement, *all* rows of the target table are deleted, as in this example:
  Delete all orders.
  DELETE FROM ORDERS
  30 rows deleted.
- Although this DELETE statement produces an empty table, it does not erase the ORDERS table from the database. The definition of the ORDERS table and its columns is still stored in the database. The table still exists, and new rows can still be inserted into the ORDERS table with the INSERT statement. To erase the definition of the table from the database, the DROP TABLE statement (described in Chapter 13) must be used.
- Because of the potential damage from such a DELETE statement, it's important always to specify a search condition and to be careful that it actually selects the rows you want.
- When using interactive SQL, it's a good idea first to use the WHERE clause in a SELECT statement to display the selected rows, make sure that they are the ones you want to delete, and only then use the WHERE clause in a DELETE statement.
- **DELETE with Subquery ***
  DELETE statements with simple search conditions, such as those in the previous examples, select rows for deletion based solely on the contents of the rows themselves. Sometimes the selection of rows must be made based on data from other tables. For example, suppose you want to delete all orders taken by Sue Smith. Without knowing her employee number, you can't find the orders by consulting the ORDERS table alone. To find the orders, you could use a two-table query:
  Find the orders taken by Sue Smith.
  SELECT ORDER_NUM, AMOUNT
  FROM ORDERS, SALESREPS
  WHERE REP = EMPL_NUM
  AND NAME = 'Sue Smith'
  ORDER_NUM AMOUNT
  ---------- ----------
  112979 $15,000.00
  113065 $2,130.00
  112993 $1,896.00
  113048 $3,750.00
  But you can't use a join in a DELETE statement. The parallel DELETE statement is illegal:
  DELETE FROM ORDERS, SALESREPS
  WHERE REP = EMPL_NUM
  AND NAME = 'Sue Smith'
  Error: More than one table specified in FROM clause
- The way to handle the request is with one of the *subquery* search conditions. Here is a valid form of the DELETE statement that handles the request:
  Delete the orders taken by Sue Smith.
  DELETE FROM ORDERS
  WHERE REP = (SELECT EMPL_NUM
  FROM SALESREPS
  WHERE NAME = 'Sue Smith')
  4 rows deleted.
- The subquery finds the employee number for Sue Smith, and the WHERE clause then selects the orders with a matching value. As this example shows, subqueries can play an important role in the DELETE statement because they let you delete rows based on information in other tables. Here are two more examples of DELETE statements that use subquery search conditions:
  Delete customers served by salespeople whose sales are less than 80 percent of quota.

```
DELETE FROM CUSTOMERS
WHERE CUST_REP IN (SELECT EMPL_NUM
FROM SALESREPS
WHERE SALES < (.8 * QUOTA))
2 rows deleted.
```

Delete any salesperson whose current orders total less than 2 percent of their quota.

```
DELETE FROM SALESREPS
WHERE (.02 * QUOTA) > (SELECT SUM(AMOUNT)
FROM ORDERS
WHERE REP = EMPL_NUM)
1 row deleted.
```

- Subqueries in the WHERE clause can be nested just as they can be in the WHERE clause of the SELECT statement. They can also contain outer references to the target table of the DELETE statement. In this respect, the FROM clause of the DELETE statement functions like the FROM clause of the SELECT statement. Here is an example of a deletion request that requires a subquery with an outer reference:

  Delete customers who have not ordered since November 10, 1989.

  ```
  DELETE FROM CUSTOMERS
  WHERE NOT EXISTS (SELECT *
  FROM ORDERS
  WHERE CUST = CUST_NUM
  AND ORDER_DATE < '10-NOV-89')
  16 rows deleted.
  ```

- Conceptually, this DELETE statement operates by going through the CUSTOMERS table, row by row, and checking the search condition. For each customer, the subquery selects any orders placed by that customer before the cutoff date. The reference to the CUST_NUM column in the subquery is an outer reference to the customer number in the row of the CUSTOMERS table currently being checked by the DELETE statement. The subquery in this example is a correlated subquery, as described in Chapter 9.
- Outer references will often be found in subqueries of a DELETE statement, because they implement the "join" between the table(s) in the subquery and the target table of the DELETE statement. In the SQL1 standard, a restriction on the use of subqueries in a DELETE statement prevented the target table from appearing in the FROM clause of a subquery or any of its subqueries at any level of nesting. This prevents the subqueries from referencing the target table (some of whose rows may already have been deleted), except for outer references to the row currently being tested by the DELETE statement's search condition. The SQL2 standard eliminated this restriction by specifying that the DELETE statement should treat such a subquery as applying to the entire target table, before any rows have been deleted. This places more overhead on the DBMS (which must handle the subquery processing and row deletion more carefully), but the behavior of the statement is well defined by the standard.

Quick Revision

- The single-row INSERT statement adds one row of data to a table. The values for the new row are specified in the statement as constants.
- The multi-row INSERT statement adds zero or more rows to a table. The values for the new rows come from a query, specified as part of the INSERT statement.
- The DELETE statement deletes zero or more rows of data from a table. The rows to be deleted are specified by a search condition.
- The UPDATE statement modifies the values of one or more columns in zero or more rows of a table. The rows to be updated are specified by a search condition.
- The columns to be updated, and the expressions that calculate their new values, are specified in the UPDATE statement.
- Unlike the SELECT statement, which can operate on multiple tables, the INSERT, DELETE, and UPDATE statements work on only a single table at a time.
- The search condition used in the DELETE and UPDATE statements has the same form as the search condition for the SELECT statement.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Write a short note on SELECT statement?	16
2	Q. Write a short note on FROM Clause?	17
3	Q. Write a short note on WHERE clause?	18
4	Q. Write a short note on ORDER By and GROUP BY?	20
5	Q. Write a short note on INSERT statement?	22
6	Q. Write a short note on UPDATE statement?	25
7	Q. Write a short note on DELETE Statement?	28

CHAPTER 4: FUNCTIONS

Topic covered:

- Aggregate functions-AVG, SUM, MIN, MAX and COUNT,
- Date functions-DATEADD(),DATEDIFF(),GETDATE(),DATENAME()YEAR,MONTH,WEEK, DAY,
- String functions- LOWER(), UPPER(), TRIM(), RTRIM(),PATINDEX(), REPLICATE(), REVERSE(),RIGHT(), LEFT()

Q. Write a short note on Aggregate Functions?

- **Computing a Column Total (SUM)**
  - The SUM() column function computes the sum of a column of data values. The data in the column must have a numeric type (integer, decimal, floating point, or money). The result of the SUM() function has the same basic data type as the data in the column, but the result may have a higher precision. For example, if you apply the SUM() function to a column of 16-bit integers, it may produce a 32-bit integer as its result.
    Here are some examples that use the SUM() column function:
    What are the total quotas and sales for all salespeople?

```
SELECT SUM(QUOTA), SUM(SALES)
FROM SALESREPS
SUM(QUOTA) SUM(SALES)
---------- ----------
$2,700,000.00 $2,893,532.00
```

    What is the total of the orders taken by Bill Adams?

```
SELECT SUM(AMOUNT)
FROM ORDERS, SALESREPS
WHERE NAME = 'Bill Adams'
AND REP = EMPL_NUM
SUM(AMOUNT)
------------
$39,327.00
```

- **Computing a Column Average (AVG)**
  - The AVG() column function computes the average of a column of data values. As with the SUM() function, the data in the column must have a numeric type. Because the AVG() function adds the values in the column and then divides by the number of values, its result may have a different data type than that of the values in the column. For example, if you apply the AVG() function to a column of integers, the result will be either a decimal or a floating point number, depending on the brand of DBMS you are using.
    Here are some examples of the AVG() column function:
    Calculate the average price of products from manufacturer ACI.

```
SELECT AVG(PRICE)
FROM PRODUCTS
WHERE MFR_ID = 'ACI'
AVG(PRICE)
-----------
$804.29
```

    Calculate the average size of an order placed by Acme Mfg. (customer number 2103).

```
SELECT AVG(AMOUNT)
FROM ORDERS
WHERE CUST = 2103
AVG(AMOUNT)
------------
$8,895.50
```

- **Finding Extreme Values (MIN and MAX)**
  - The MIN() and MAX() column functions find the smallest and largest values in a column,

respectively. The data in the column can contain numeric, string, or date/time information.

- The result of the MIN() or MAX() function has exactly the same data type as the data in the column. Here are some examples that show the use of these column functions:
  What are the smallest and largest assigned quotas?
  SELECT MIN(QUOTA), MAX(QUOTA)
  FROM SALESREPS
  MIN(QUOTA) MAX(QUOTA)
  ---------- ----------
  $200,000.00 $350,000.00
  What is the earliest order date in the database?
  SELECT MIN(ORDER_DATE)
  FROM ORDERS
  MIN(ORDER_DATE)

  04-JAN-89
  What is the best sales performance of any salesperson?
  SELECT MAX(100 * (SALES/QUOTA))
  FROM SALESREPS
  MAX(100*(SALES/QUOTA))

  135.44
- When the MIN() and MAX() column functions are applied to numeric data, SQL compares the numbers in algebraic order (large negative numbers are less than small negative numbers, which are less than zero, which is less than all positive numbers). Dates are compared sequentially (earlier dates are smaller than later ones). Durations are compared based on their length (shorter durations are smaller than longer ones).
- When using MIN() and MAX() with string data, the comparison of two strings depends upon the character set being used. On a personal computer or minicomputer, both of which use the ASCII character set, digits come before the letters in the sorting sequence, and all of the uppercase characters come before all of the lowercase characters. On IBM mainframes, which use the EBCDIC character set, the lowercase characters precede the uppercase characters, and digits come after the letters. Here is a comparison of the ASCII and EBCDIC collating sequences of a list of strings, from smallest to largest:
  - **ASCII**
  - **EBCDIC**
  - 1234ABC
  - a cme mfg.
  - 5678ABC
  - z eta corp.
  - ACME MFG.
  - A cme Mfg.
  - Acme Mfg.
  - A CME MFG.
  - ZETA CORP.
  - Z eta Corp.
  - Zeta Corp.
  - Z ETA CORP.
  - acme mfg.
  - 1 234ABC
  - zeta corp.
  - 5 678ABC
- The difference in the collating sequences means that a query with an ORDER BY clause can produce different results on two different systems.
  International characters (for example, accented characters in French, German, Spanish, or Italian or the Cyrillic alphabet letters used in Greek or Russian, or the Kanji symbols used in Japanese) pose additional problems. Some brands of DBMS use special

international sorting algorithms to sort these characters into their correct position for each language. Others simply sort them according to the numeric value of the code assigned to the character. To address these issues, the SQL2 standard includes elaborate support for national character sets, user-defined character sets, and alternate collating sequences. Unfortunately, support for these SQL2 features varies widely among popular DBMS products. If your application involves international text, you will want to experiment with your particular DBMS to find out how it handles these characters.

- **Counting Data Values (COUNT)**

- The COUNT() column function counts the number of data values in a column. The data in the column can be of any type. The COUNT() function always returns an integer, regardless of the data type of the column. Here are some examples of queries that use the COUNT() column function:

```
How many customers are there?
SELECT COUNT(CUST_NUM)
FROM CUSTOMERS
COUNT(CUST_NUM)
---------------
21
How many salespeople are over quota?
SELECT COUNT(NAME)
FROM SALESREPS
WHERE SALES > QUOTA
COUNT(NAME)
------------
7
How many orders for more than $25,000 are on the books?
SELECT COUNT(AMOUNT)
FROM ORDERS
WHERE AMOUNT > 25000.00
COUNT(AMOUNT)
--------------
4
```

- Note that the COUNT() function ignores the values of the data items in the column; it simply counts how many data items there are. As a result, it doesn't really matter which column you specify as the argument of the COUNT() function. The last example could just as well have been written this way:

```
SELECT COUNT(ORDER_NUM)
FROM ORDERS
WHERE AMOUNT > 25000.00
COUNT(ORDER_NUM)
-----------------
4
```

- In fact, it's awkward to think of the query as "counting how many order amounts" or "counting how many order numbers;" it's much easier to think about "counting how many orders." For this reason, SQL supports a special COUNT(*) column function, which counts rows rather than data values. Here is the same query, rewritten once again to use the COUNT(*) function:

```
SELECT COUNT(*)
FROM ORDERS
WHERE AMOUNT > 25000.00
COUNT(*)
---------
4
```

- If you think of the COUNT(*) function as a "rowcount" function, it makes the query easier to read. In practice, the COUNT(*) function is almost always used instead of the COUNT() function to count rows.

Q. Write a short note on Date Functions?

- **The DATEADD() function is adds or subtracts a specified time interval from a date.**
- **Syntax**

DATEADD(datepart,number,date)

- Where date is a valid date expression and number is the number of interval you want to add. The number can either be positive, for dates in the future, or negative, for dates in the past.

  datepart can be one of the following:

DATEDIFF and DATEADD

DATEDIFF and DATEADD both work on the basis of determining the difference between two dates in a given unit of time. With DATEDIFF, you provide the two dates and the return value will be the difference in the unit you specify; with DATEADD you provide the first date and the difference, and the return value will be the second date.

DATEDIFF

The DATEDIFF function determines the difference (the 'interval') between two date values. The function takes three arguments, as follows:

DATEDIFF (interval, date1, date2)

The first parameter is any valid datepart value, as listed in the section on DATENAME and DATPART here. This determines the unit in which the difference should be calculated. The second and third parameters are the date values to be compared. If date1 is before date2, the result will be a positive value, if date1 is after date2 the result will be a negative value.

Let's take a look at an example of the output for various parameter values. In the following script, the first two columns extract the OrderDate and the ShippedDate values from the Orders table. The third column expresses the difference between these two dates in days, the fourth expresses it in weeks.

SELECT OrderDate, ShippedDate,

DATEDIFF(d,OrderDate, ShippedDate) as 'Ship days',

DATEDIFF(wk, OrderDate, ShippedDate) as 'Ship weeks'

FROM Orders

Here's the output from this query:

	OrderDate	ShippedDate	Ship days	Ship weeks
1	1996-07-04 00:00:00.000	1996-07-16 00:00:00.000	12	2
2	1996-07-05 00:00:00.000	1996-07-10 00:00:00.000	5	1
3	1996-07-08 00:00:00.000	1996-07-12 00:00:00.000	4	0
4	1996-07-08 00:00:00.000	1996-07-15 00:00:00.000	7	1
5	1996-07-09 00:00:00.000	1996-07-11 00:00:00.000	2	0
6	1996-07-10 00:00:00.000	1996-07-16 00:00:00.000	6	1
7	1996-07-11 00:00:00.000	1996-07-23 00:00:00.000	12	2
8	1996-07-12 00:00:00.000	1996-07-15 00:00:00.000	3	1
9	1996-07-15 00:00:00.000	1996-07-17 00:00:00.000	2	0
10	1996-07-16 00:00:00.000	1996-07-22 00:00:00.000	6	1
11	1996-07-17 00:00:00.000	1996-07-23 00:00:00.000	6	1

In the next example, we are calculating the age of each of the employees listed in the Employees table. We are using DATEDIFF to calculate the difference in years between the BirthDate field and the current date as returned by the GETDATE function. Heres the SQL statement:

SELECT FirstName +' ' +LastName as 'Name',

Birthdate,

DATEDIFF(yy, Birthdate, GETDATE()) as 'Age'

FROM Employees

And here's the output:

	Name	Birthdate	Age
1	Nancy Davolio	1948-12-08 00:00:00.000	60
2	Andrew Fuller	1952-02-19 00:00:00.000	56
3	Janet Leverling	1963-08-30 00:00:00.000	45
4	Margaret Peacock	1937-09-19 00:00:00.000	71
5	Steven Buchanan	1955-03-04 00:00:00.000	53
6	Michael Suyama	1963-07-02 00:00:00.000	45
7	Robert King	1960-05-29 00:00:00.000	48
8	Laura Callahan	1958-01-09 00:00:00.000	50
9	Anne Dodsworth	1966-01-27 00:00:00.000	42

If we look carefully at this output, there appear to be some anomalies. The date on which this query was run was 21 May 2008... and yet Nancy Davolio is calculated as being 60, despite the fact that her birthday is not until December. Why is this? It's certainly not rounding, as she's actually closer to her 59th birthday than her 60th.

In fact DATEDIFF calculates *how many boundaries have been crossed* of the specified type. So, given that we're looking for years in this example, the first boundary in the first row is crossed on December 31 1947 - only 23 days after Nancy's date of birth. And we've crossed another one each year, including one on 31 December 2007 - the 60th boundary to be crossed in the 59 years of her birth.

You may therefore find it to be a more accurate measure to use **dd** as the interval parameter, and then divide the result by 365.25 in order to determine the number of years. This will result in a decimal being

returned, so the FLOOR function may be used to round this value down to the nearest whole number, as follows:

SELECT FIRSTNAME + ' ' + LASTNAME as 'Name',

BIRTHDATE,

DATEDIFF(dd,BIRTHDATE,GETDATE())/365.25 as 'Exact Age',

FLOOR(DATE(dd,DATEDIFF,DATEDIFF())/365.25) as 'Age'

FROM Employees

	Name	BirthDate	Exact Age	Age
1	Nancy Davolio	1948-12-08 00:00:00.000	59.452429	59
2	Andrew Fuller	1952-02-19 00:00:00.000	56.254620	56
3	Janet Leverling	1963-08-30 00:00:00.000	44.728268	44
4	Margaret Peacock	1937-09-19 00:00:00.000	70.672142	70
5	Steven Buchanan	1955-03-04 00:00:00.000	53.218343	53
6	Michael Suyama	1963-07-02 00:00:00.000	44.889801	44
7	Robert King	1960-05-29 00:00:00.000	47.980835	47
8	Laura Callahan	1958-01-09 00:00:00.000	50.365503	50
9	Anne Dodsworth	1966-01-27 00:00:00.000	42.316221	42

DATEADD

DATEADD is used when an interval is to be added to a given date in order to return a second date. The values allowed for the interval argument are identical to those for the DATEDIFF function. The full syntax is as follows:

DATEADD(Interval, Number, Date)

As an example, consider a requirement that all orders are shipped within seven days. The following statement shows the date seven days after the order date for each order in the orders table:

SELECT OrderID, OrderDate,

DATEADD(dd,7,Orderdate) as 'Ship date reqd'

FROM Orders

This generates the following output:

	OrderID	OrderDate	Ship date reqd
1	10248	1996-07-04 00:00:00.000	1996-07-11 00:00:00.000
2	10249	1996-07-05 00:00:00.000	1996-07-12 00:00:00.000
3	10250	1996-07-08 00:00:00.000	1996-07-15 00:00:00.000
4	10251	1996-07-08 00:00:00.000	1996-07-15 00:00:00.000
5	10252	1996-07-09 00:00:00.000	1996-07-16 00:00:00.000
6	10253	1996-07-10 00:00:00.000	1996-07-17 00:00:00.000
7	10254	1996-07-11 00:00:00.000	1996-07-18 00:00:00.000
8	10255	1996-07-12 00:00:00.000	1996-07-19 00:00:00.000
9	10256	1996-07-15 00:00:00.000	1996-07-22 00:00:00.000
10	10257	1996-07-16 00:00:00.000	1996-07-23 00:00:00.000

SQL Server Date Functions

The following table lists the most important built-in date functions in SQL Server:

Function	Description
GETDATE()	Returns the current date and time
DATEPART()	Returns a single part of a date/time
DATEADD()	Adds or subtracts a specified time interval from a date
DATEDIFF()	Returns the time between two dates
CONVERT()	Displays date/time data in different formats

GETDATE ()
The **GETDATE ()** function returns the current date and time from the server's system clock. This function can be used to automatically insert date values into columns or to find out the current date for comparisons in WHERE clauses. Audit trails and tracking tables benefit greatly from this function and the **CURRENT_USER** function.

DATEDIFF()
You can use this function to compare and return the difference between date items such as days, weeks, minutes, and hours. When this function is used in a WHERE clause, you can return records that meet a range of dates, or that meet certain time-span intervals. Make sure you specify the arguments to this function in the correct order, or the value returned might be the opposite of what you expected.

DATEPART()
This function returns a value equal to the part of a date that you specify. If, for instance, you need to know the day of the week of a particular date, you can use this function to quickly pull that data out of a column or variable.

DATENAME ()

It return specifies the part of the date name and the DATENAME for Date Time such as Year, Quarter, Month, Day, Hour, Minute and Millisecond.

SQL DATENAME Syntax
DATENAME (datepart , date)

DATEPART and DATENAME

Note: These functions do not exist in Access (JET SQL) . For information on equivalents in Access SQL, see Access Date Functions.

The datename and datepart functions are used within T-SQL for extracting information about a date, in much the same way as the date functions discussed on the previous page. They are, however, slightly more complex, in that unlike DATE or MONTH or YEAR they require more than one parameter.

Both functions are similar in their construction - they take parameters indicating the type of information required and the date to be analysed. The difference between them is that DATENAME returns a string indicating a portion of a date - such as 'Monday' or 'March' whereas DATEPART returns an integer, such as 2 for Monday or 3 for March. In many instances (for example when returning the year) both functions will return a number. However, in all cases DATEPART returns an integer, whereas DATENAME returns a string - specifically the T-SQL nvarchar data type.

Perhaps the hardest part is remembering the values for the first parameter - that which indicates which portion of the date to return. The table below (taken from SQL Server help) gives this information, and is followed by some examples of using both DATEPART and DATENAME.

Datepart	Abbreviations
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

Values for the first argument of DATENAME and DATEPART.

So, given that GETDATE() returns the current date and time, which at the time of writing is Thursday, 22 May 2008 at 07:59:23.347, lets take a look at the output from these functions. In the first column in the table below is the value for the first argument of either function, which may be replaced by any of the abbreviations in the table above. The second column displays the return value from the DATEPART function, which would be written as follows (for the first example):

SELECT DATEPART(year, GETDATE())

The third column displays the return value from the DATENAME function, which would be written as follows, again for the first example:

SELECT DATENAME (year, GETDATE())

First argument	DATEPART return	DATENAME return
year	2008	2008
quarter	2	2
month	5	May
dayofyear	143	143
day	22	22
week	21	21
weekday	5	Thursday
hour	7	7
minute	59	59
second	23	23
millisecond	347	347

Return values from DATEPART and DATENAME

Uses of DATEPART and DATENAME

Given that in many cases DATEPART and DATENAME seem to return identical values, when is it necessary to use one function rather than the other?

Firstly, there are the obvious cases of using **month** and **weekday** as the first parameter - if the name of the day or month is required, then obviously you would use DATENAME. However, this can pose problems when the output needs to be sorted by month - if an ORDER BY clause is followed by the DATENAME function, the output will be sorted in alphabetical order, making April the first month, followed by August, then December and so on. In this instance, it is often necessary to include the DATENAME function in the select list, but use the DATEPART function in the ORDER BY clause to ensure proper sorting.

Beyond that, the key lies in the data type being returned. As mentioned above, the return value from DATEPART is always **int** whereas from DATENAME it is always **nvarchar**. There will be occasions when, for example, creating stored procedures (which is beyond the scope of this article, but may be a topic for a future article) when you want to return a numeric value rather than a varchar, or vice versa. In such cases the use of the appropriate function here will negate the need for use of the CONVERT function to convert data from one type to another.

MONTH(date) *or* DATEPART(m, date) ?

In the previous section, we discussed the MONTH, DAY and YEAR functions. So, which should you use - DATEPART with an appropriate first parameter, or MONTH, DAY and YEAR? In short, it doesn't matter - the two functions are synonymous when returning integers representing the current day of the month, month number or year.

Do bear in mind, however, that DATENAME and DATEPART do not exist in Microsoft Access. Therefore if you want to write a script which will work in both Access and SQL Server, you will need to use DAY, MONTH and YEAR in preference to DATEPART.

GETDATE, DAY, MONTH and YEAR

There are a number of functions within SQL for processing dates. These include functions for determining the difference between two dates, for returning information about a date and for adding a value to a date to return a new date.

On this page, we will take a look at some of the basic date functions - GETDATE, DAY, WEEKDAY, MONTH and YEAR. Over the next pages we will look at DATEPART, DATENAME, DATEDIFF and DATEADD.

GETDATE

GETDATE is an unusual function in the sense that it does not process a given date, or indeed need any other input parameter. It simply returns the current system date and time. Its syntax is simple:

```
SELECT GETDATE ()
```

Note that the use of the brackets to indicate the use of a function is required - an error will be returned if the brackets are omitted. The value returned is of the DATETIME data type and will include the date and the time.

GETDATE is frequently used when comparing a given date to the current date, for example in determining how long ago an order date was, or how far in the future a shipment date is. For more information on accomplishing these tasks, see the section on DATEDIFF.

DAY, MONTH and YEAR

These functions are similar in their functionality. Each takes a single input parameter and returns an integer.

DAY returns a value between 1 and 31 indicating the day of the month for the given date.

MONTH returns a value between 1 and 12 indicating the month of a given date.

YEAR returns the year portion of the given date.

The following code allows us to see the output of each of these functions:

```
SELECT orderdate,

DAY(orderdate) as 'Day',

MONTH(orderdate) as 'Month',

YEAR(orderdate) as 'Year'

FROM orders
```

This returns the following:

	orderdate	Day	Month	Year
1	1996-07-04 00:00:00.000	4	7	1996
2	1996-07-05 00:00:00.000	5	7	1996
3	1996-07-08 00:00:00.000	8	7	1996
4	1996-07-08 00:00:00.000	8	7	1996
5	1996-07-09 00:00:00.000	9	7	1996
6	1996-07-10 00:00:00.000	10	7	1996
7	1996-07-11 00:00:00.000	11	7	1996
8	1996-07-12 00:00:00.000	12	7	1996
9	1996-07-15 00:00:00.000	15	7	1996
10	1996-07-16 00:00:00.000	16	7	1996

In the next section, we will see the use of DATEPART and DATENAME. DATEPART is able to return values which are synonymous with DAY, MONTH and YEAR, but as we shall see, it also has additional capabilities.

Q. Write a short note on String functions?

- **LOWER():**
  - **LOWER** string function returns the lower case string whether the entered string has upper case letters. It takes 1 argument as string value.

- **Example:**
  select LOWER('MICROSOFT ASP .NET WEB HOSTING')
  Result: microsoft asp .net web hosting
- **LOWER:** Convert character strings data into lowercase.
- Syntax: LOWER(string)
- SELECT LOWER ('STRING FUNCTION')->string function

➢ **UPPER():**

- **UPPER()** converts the string passed to the function into all uppercase characters. You can use this function to maintain the data integrity of text columns in your tables without the user or client intervening. Some client software always assumes this function to be bound to a control with very little overhead, so I seldom use this for much more than making sure I pass uppercase characters where needed.
- **UPPER** string function returns the upper case string whether the entered string has lower case letters. It takes 1 argument as string value.
- **Example:**
  select LOWER('MICROSOFT ASP .NET WEB HOSTING with SQL Database')
  Result: MICROSOFT ASP .NET WEB HOSTING WITH SQL DATABASE
- **UPPER:** Convert character strings data into Uppercase.
- Syntax: UPPER (string)
- SELECT UPPER ('string function')->STRING FUNCTION

➢ **LTRIM():**

- **LTRIM** function returns the string by removing all the blank spaces at left side. It also takes 1 argument as string value.
- **Example:**
  select LTRIM (' ASP ')
  Result: ASP-----
  blanks at the right side not removed.
- **LTRIM :** Returns a string after removing leading blanks on Left side.(Remove left side space or blanks)
- Syntax: LTRIM(string)
- SELECT LTRIM(' STRING FUNCTION')->STRING FUNCTION

➢ **RTRIM():**

- **RTRIM :** Returns a string after removing leading blanks on Right side.(Remove right side space or blanks)
- Syntax: RTRIM(string)
- SELECT RTRIM('STRING FUNCTION ')->STRING FUNCTION
- **RTRIM** function returns the string by removing all the blank spaces at left side. It also takes 1 argument as string value.
- **Example:**
  select RTRIM (' ASP ')
  Result: -----ASP
  blanks at the left side not removed.

➢ **PATINDEX():**

- **PATINDEX** function returns the position of first occurrence of specified pattern in the provided string. It takes 2 arguments.
  1st argument as string value specifying the pattern to match
  2nd argument as string value specifying the string to compare.
- **Example:**
  select PATINDEX('%RO%','MICROSOFT')
  Results: 4
- PATINDEX function works very similar to CHARINDEX function.PATINDEX function returns the starting position of the first occurrence of a pattern in a specified string, or zeros if the pattern is not found.

- Using PATINDEX function you can search pattern in given string using Wildcard characters(%).The % character must come before and after pattern.
- Syntax: PATINDEX('%pattern%',string)
- SELECT PATINDEX('%SQL%','Useful SQL String Function')->8
- SELECT PATINDEX('Useful%','Useful SQL String Function')->1
- SELECT PATINDEX('%Function','Useful SQL String Function')->19
- If pattern is not found within given string,PATINDEX returns 0.
- Repeats a character expression for a specified number of times.

- **Syntax**
- REPLICATE **(** *character_expression* **,** *integer_expression* **)**

- Arguments
- *character_expression*
- Is an alphanumeric expression of character data. *character_expression* can be a constant, variable, or column of either character or binary data.
- *integer_expression*
- Is a positive whole number. If *integer_expression* is negative, a null string is returned.

- Return Types
- **varchar**
- *character_expression* must be of a data type that is implicitly convertible to **varchar**. Otherwise, use the CAST function to convert explicitly *character_expression*.

➢ **REPLICATE():**
- **REPLICATE :** Repeats a input string for a specified number of times.
- Syntax: REPLICATE (string, integer)
- SELECT REPLICATE ('FUNCTION', 3)->FUNCTIONFUNCTIONFUNCTION

- *Use REPLICATE*
- This example replicates each author's first name twice.
- USE pubs
- SELECT REPLICATE (au_fname, 2)
- FROM authors
- ORDER BY au_fname
- Here is the result set:

```
----------------------
AbrahamAbraham
AkikoAkiko
AlbertAlbert
AnnAnn
AnneAnne
BurtBurt
CharleneCharlene
CherylCheryl
DeanDean
DirkDirk
HeatherHeather
InnesInnes
(12row(s) affected)
```

➢ **REVERSE():**
- **REVERSE** string function returns the string in reverse order. It takes 1 argument as string value.
  Example:
  select REVERSE('ASP.NET')
  Result: TEN.PSA
- **REVERSE:** Returns reverse of a input string.

- Syntax: REVERSE(string)
- SELECT REVERSE('STRING FUNCTION')->NOITCNUF GNIRTS

- **RIGHT():**
  - **RIGHT** string function takes 2 arguments. 1st argument takes as a string value and 2nd argument as integer value as length parameter. It returns last characters of specified length starting from the right side of the string entered as 1st argument.
  - **Example:**
    Select RIGHT ('MICROSOFT SQL SERVER 2000',4)
    Result: 2000
  - **RIGHT:** Returns right part of a string with the specified number of characters counting from right. RIGHT function is used to retrieve portions of the string.
  - Syntax: RIGHT(string,integer)
  - SELECT RIGHT('STRING FUNCTION', 8)->FUNCTION
- **LEFT ():**
  - **LEFT** string function takes 2 arguments. 1st argument takes as a string value and 2nd argument as integer value as length parameter. It returns first characters of specified length starting from the left side of the string entered as 1st argument.
  - **Example:**
    Select LEFT ('MICROSOFT SQL SERVER 2000',4)
    Result: MICR
  - **LEFT:** Returns left part of a string with the specified number of characters counting from left. LEFT function is used to retrieve portions of the string.
  - Syntax: LEFT(string,integer)
  - SELECT LEFT('STRING FUNCTION', 6)->STRING

Quick Revision

- The SUM() column function computes the sum of a column of data values.
- The AVG() column function computes the average of a column of data values.
- The MIN() and MAX() column functions find the smallest and largest values in a column.
- The COUNT() column function counts the number of data values in a column.
- The DATEADD() function is adds or subtracts a specified time interval from a date.
- The DATEDIFF function determines the difference (the 'interval') between two date values.
- The **GETDATE ()** function returns the current date and time from the server's system clock.
- **LOWER** string function returns the lower case string whether the entered string has upper case letters. It takes 1 argument as string value.
- **UPPER ()** converts the string passed to the function into all uppercase characters.

- **LTRIM** function returns the string by removing all the blank spaces at left side. It also takes 1 argument as string value.
- **RTRIM :** Returns a string after removing leading blanks on Right side.(Remove right side space or blanks)
- **PATINDEX** function returns the position of first occurrence of specified pattern in the provided string. It takes 2 arguments.
- **REPLICATE:** Repeats a input string for a specified number of times.
- **REVERSE** string function returns the string in reverse order. It takes 1 argument as string value.
- **RIGHT** string function takes 2 arguments. 1st argument takes as a string value and 2nd argument as integer value as length parameter.
- **LEFT** string function takes 2 arguments. 1st argument takes as a string value and 2nd argument as integer value as length parameter.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Write a short note on Aggregate Functions?	32
2	Q. Write a short note on Date Functions?	35
3	Q. Write a short note on String functions?	42

CHAPTER 5: JOINING TABELS

Topic covered:

- Inner, outer and cross joins, union.

Q. Write a short note on Inner joins?

- **Inner Joins in SQL2 ***
  - Figure 7-13 shows a simplified form of the extended SQL2 syntax for the FROM clause. It's easiest to understand all of the options provided by considering each type of join, one by one, starting with the basic inner join and then moving to the various forms of outer join. The standard inner join of the GIRLS and BOYS tables can be expressed in SQL1 language:

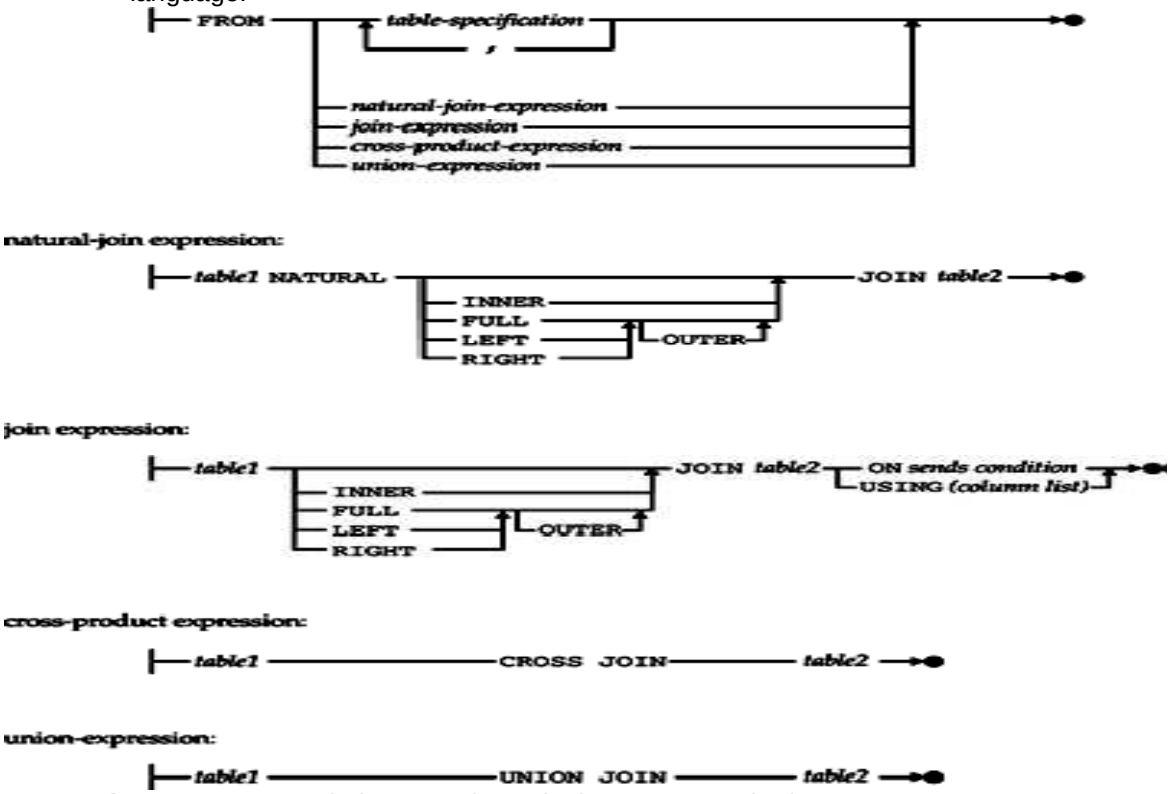


Figure 7-13: Extended FROM clause in the SQL2 standard

```
SELECT *
FROM GIRLS, BOYS
WHERE GIRLS.CITY = BOYS.CITY
```

- This is still an acceptable statement in SQL2. The writers of the SQL2 standard really couldn't have made it illegal without "breaking" all of the millions of multi-table SQL queries that had already been written by the early 1990s. But the SQL2 standard specifies an alternative way of expressing the same query:

```
SELECT *
FROM GIRLS INNER JOIN BOYS
ON GIRLS.CITY = BOYS.CITY
```

- Note that the two tables to be joined are explicitly connected by a JOIN operation, and the search condition that describes the join is now specified in an ON clause within the FROM clause. The search condition following the keyword ON can be any search condition that specifies the criteria used to match rows of the two joined tables. The columns referenced in the search condition must come only from the two joined tables. For example, assume that the BOYS table and the GIRLS table were each extended by adding an AGE column. Here is a join that matches girl/boy pairs in the same city and also requires that the boy and girl in each pair be the same age:

```
SELECT *
FROM GIRLS INNER JOIN BOYS
ON (GIRLS.CITY = BOYS.CITY)
AND (GIRLS.AGE = BOYS.AGE)
```

- In these simple two-table joins, the entire contents of the WHERE clause simply moved into the ON clause, and the ON clause doesn't add any functionality to the SQL language.
- However, recall from earlier in this chapter that in a outer join involving three tables or more, the order in which the joins occur affect the query results. The ON clause provides detailed control over how these multi-table joins are processed, as described later in this chapter.
- The SQL2 standard permits another variation on the simple inner join query between the GIRLS and BOYS tables. Because the matching columns in the two tables have the same names and are being compared for equality (which is often the case), an alternative form of the ON clause, specifying a list of matching column names, can be used:

```
SELECT *
FROM GIRLS INNER JOIN BOYS
USING (CITY, AGE)
```

- The USING clause specifies a comma-separated list of the matching column names, which must be identical in both tables. It is completely equivalent to the ON clause that specifies each matching column pair explicitly, but it's a lot more compact and therefore easier to understand. Of course, if the matching columns have different names in the BOYS table and GIRLS table, then an ON clause or a WHERE clause with an equals test must be used. The ON clause must also be used if the join does not involve equality of the matching columns. For example, if you wanted to select girl/boy pairs where the girl was required to be older than the boy, you must use an ON clause to specify the join:

```
SELECT *
FROM GIRLS INNER JOIN BOYS
ON (GIRLS.CITY = BOYS.CITY
AND GIRLS.AGE > BOYS.AGE)
```

- There is one final variation on this simple query that illustrates another feature of the SQL2 FROM clause. A join between two tables where the matching columns are *exactly* those specific columns from the two tables that have identical names is called a *natural join,* because usually this is precisely the most "natural" way to join the two tables. The query selecting girl/boy pairs who live in the same city and have the same age can be expressed as a natural join using this SQL2 query:

```
SELECT *
FROM GIRLS NATURAL INNER JOIN BOYS
```

- If the NATURAL keyword is specified, the ON and USING clauses may not be used in the join specification, because the natural join specifically defines the search condition to be used to join the tables—all of the columns with identical column names in both tables are to be matched.
- The SQL2 standard assumes that the "default" join between two tables is an inner join. You can omit the keyword INNER from any of the preceding examples, and the resulting query remains a legal SQL2 statement with the same meaning.

Q. Write a short note on outer joins?

➢ **Outer Joins in SQL2 ***

- The SQL2 standard provides complete support for outer joins using the same clauses described in the preceding section for inner joins and additional keywords. For example, the full outer join of the GIRLS and BOYS tables (without the AGE columns) is generated by this query:

```
SELECT *
FROM GIRLS FULL OUTER JOIN BOYS
ON GIRLS.CITY = BOYS.CITY
```

- As explained earlier in this chapter, the query results will contain a row for each matched girl/boy pair, as well as one row for each unmatched boy, extended with NULL values in the columns from the other, unmatched table. SQL2 allows the same variations for outer joins as for inner joins; the query could also have been written:

```
SELECT *
FROM GIRLS NATURAL FULL OUTER JOIN BOYS
```
or
```
SELECT *
FROM GIRLS FULL OUTER JOIN BOYS
USING (CITY)
```
- Just as the keyword INNER is optional in the SQL2 language, the SQL2 standard also allows you to omit the keyword OUTER. The preceding query could also have been written:

```
SELECT *
FROM GIRLS FULL JOIN BOYS
USING (CITY)
```
The DBMS can infer from the word FULL that an outer join is required.
- By specifying LEFT or RIGHT instead of FULL, the SQL2 language extends quite naturally to left or right outer joins. Here is the left outer join version of the same query:

```
SELECT *
FROM GIRLS LEFT OUTER JOIN BOYS
USING (CITY)
```
- As described earlier in the chapter, the query results will include matched girl/boy pairs and NULL-extended rows for each unmatched row in the GIRLS table (the "left" table in the join), but the results do not include unmatched rows from the BOYS table. Conversely, the right outer join version of the same query, specified like this:

```
SELECT *
FROM GIRLS RIGHT OUTER JOIN BOYS
USING (CITY)
```
includes boy/girl pairs and unmatched rows in the BOYS table (the "right" table in the join) but does not include unmatched rows from the GIRLS table.

Q. Write a short note on cross joins and Union?

➢ **Cross Joins and Union Joins in SQL2 ***

- The SQL2 support for extended joins includes two other methods for combining data from two tables. A *cross join* is another name for the Cartesian product of two tables, as described earlier in this chapter. A *union join* is closely related to the full outer join; its query results are a subset of those generated by the full outer join.
  Here is a SQL2 query that generates the complete product of the GIRLS and BOYS tables:

```
SELECT *
FROM GIRLS CROSS JOIN BOYS
```
- By definition, the Cartesian product (also sometimes called the *cross product*, hence the name "CROSS JOIN") contains every possible pair of rows from the two tables. It "multiplies" the two tables, turning tables of, for example, three girls and two boys into a table of six (3 x 2 = 6) boy/girl pairs. No "matching columns" or "selection criteria" are associated with the cross products, so the ON clause and the USING clause are not allowed. Note that the cross join really doesn't add any new capabilities to the SQL language. Exactly the same query results can be generated with an inner join that specifies no matching columns. So the preceding query could just as well have been written as:

```
SELECT *
FROM GIRLS, BOYS
```
- The use of the keywords CROSS JOIN in the FROM clause simply makes the cross join more explicit. In most databases, the cross join of two tables by itself is of very little practical use. Its usefulness really comes as a building block for more complex query expressions that start with the cross product of two tables and then use SQL2 summary query capabilities (described in the next chapter) or SQL2 set operations to further manipulate the results.
- The union join in SQL2 combines some of the features of the UNION operation (described in the previous chapter) with some of the features of the join operations described in this chapter. Recall that the UNION operation effectively combines the rows of two tables,

which must have the same number of columns and the same data types for each corresponding column. This query, which uses a simple UNION operation:

```
SELECT *
FROM GIRLS
UNION ALL
SELECT *
FROM BOYS
```

- when applied to a three-row table of girls and a two-row table of boys yields a five-row table of query results. Each row of query results corresponds precisely to either a row of the GIRLS table or a row of the BOYS table from which it was derived. The query results have two columns, NAME and CITY, because the GIRLS and BOYS tables each have these two columns.
- The union join of the GIRLS and BOYS tables is specified by this SQL2 query:

```
SELECT *
FROM GIRLS
UNION JOIN BOYS
```

- The query results again have five rows, and again each row of results is contributed by exactly one of the rows in the GIRLS table or the BOYS table. But unlike the simple union, these query results have four columns—all of the columns of the first table *plus* all of the columns of the second table. In this aspect, the union join is like all of the other joins. For each row of query results contributed by the GIRLS table, the columns that come from the GIRLS table receive the corresponding data values; the other columns (those that come from the BOYS table) have NULL values. Similarly, for each row of query results contributed by the BOYS table, the columns that come from the BOYS table receive the corresponding data values; the other columns (this time, those that come from the GIRLS table) have NULL values.
- Another way of looking at the results of the union join is to compare them to the results of a full outer join of the GIRLS and BOYS tables. The union join results include the NULL extended rows of data from the GIRLS table *and* the NULL-extended rows of data from the BOYS table, but they do *not* include any of the rows generated by matching columns. Referring back to the definition of an outer join in Figure 7-14, the union join is produced by omitting Step 1 and following Step 2 and Step 3.

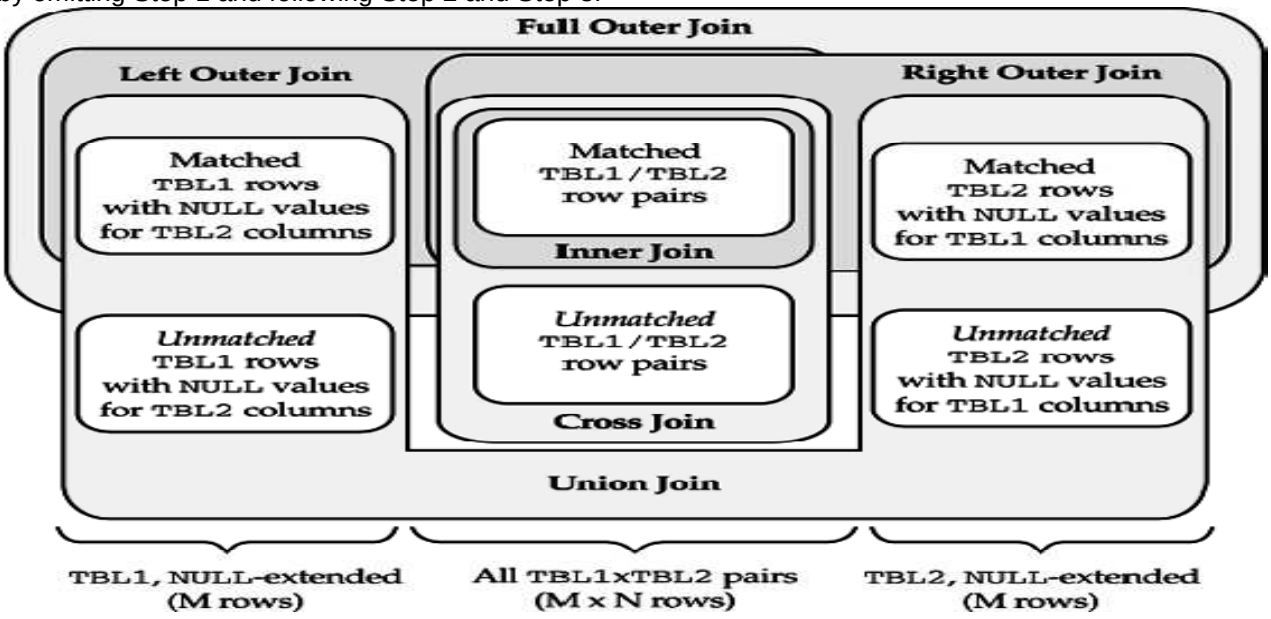


Figure 7-14: Relationships among SQL2 join types

Finally, it's useful to examine the relationships between the sets of rows produced by the cross join, the various types of outer joins, and the inner join shown in Figure 7-14. When joining two tables, TBL1 with *m* rows and TBL2 with *n* rows, the figure shows that:

- The *cross join* will contain *m* x *n* rows, consisting of all possible row pairs from the two tables.
- TBL1 INNER JOIN TBL2 will contain some number of rows, *r*, which is less than *m* x *n*. The inner join is strictly a subset of the cross join. It is formed by eliminating those rows from the cross join that do not satisfy the matching condition for the inner join.
- The *left outer join* contains all of the rows from the inner join, plus each unmatched row from TBL1, NULL-extended.
- The *right outer join* also contains all of the rows from the inner join, plus each unmatched row from TBL2, NULL-extended.
- The *full outer join* contains all of the rows from the inner join, plus each unmatched row from TBL1, NULL-extended, plus each unmatched row from TBL2, NULL-extended. Roughly speaking, its query results are equal to the left outer join "plus" the right outer join.
- The *union join* contains all of the rows of TBL1, NULL-extended, plus all of the rows of TBL2, NULL-extended. Roughly speaking, its query results are the full outer join "minus" the inner join.

Quick Revision

- In a multi-table query (a *join*), the tables containing the data are named in the FROM clause.

- Each row of query results is a combination of data from a single row in each of the tables, and it is the *only* row that draws its data from that particular combination.
- The most common multi-table queries use the parent/child relationships created by primary keys and foreign keys.
- In general, joins can be built by comparing *any* pair(s) of columns from the two joined tables, using either a test for equality or any other comparison test.
- A join can be thought of as the product of two tables from which some of the rows have been removed.
- A table can be joined to itself; self-joins require the use of a table alias.

- Outer joins extend the standard (inner) join by retaining unmatched rows of one or both of the joined tables in the query results, and using NULL values for data from the other table.

- The SQL2 standard provides comprehensive support for inner and outer joins, and for Combining the results of joins with other multi-table operations such as unions, intersections, and differences.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Write a short note on Inner joins?	47
2	Q. Write a short note on outer joins?	48
3	Q. Write a short note on cross joins and Union?	49

CHAPTER 6: SUB QUERIES

Topic Covered:

- Sub queries with IN, EXISTS, sub queries restrictions, Nested sub queries, correlated sub queries, queries with modified comparison operations.
- SELECT INTO operation, UNION operation. Sub queries in the HAVING clause.

Q. Explain Sub queries with IN and EXISTS?

- **Set Membership Test (IN)**

- The subquery set membership test (IN) is a modified form of the simple set membership test, as shown in Figure 9-4. It compares a single data value to a column of data values produced by a subquery and returns a TRUE result if the data value matches one of the values in the column. You use this test when you need to compare a value from the row being tested to a *set* of values produced by a subquery. Here is a simple example:

Figure 9-4: Subquery set membership test (IN) syntax diagram

List the salespeople who work in offices that are over target.

```
SELECT NAME
FROM SALESREPS
WHERE REP_OFFICE IN (SELECT OFFICE
FROM OFFICES
WHERE SALES > TARGET)
NAME
----------
Mary Jones
Sam Clark
Bill Adams
Sue Smith
Larry Fitch
```

- The subquery produces a set of office numbers where the sales are above target (in the sample database, there are three such offices, numbered 11, 13, and 21). The main query then checks each row of the SALESREPS table to determine whether that particular salesperson works in an office with one of these numbers. Here are some other examples of subqueries that test set membership:

List the salespeople who do not work in offices managed by Larry Fitch (employee 108).

```
SELECT NAME
FROM SALESREPS
WHERE REP_OFFICE NOT IN (SELECT OFFICE
FROM OFFICES
WHERE MGR = 108)
NAME
----------
Bill Adams
Mary Jones
Sam Clark
Bob Smith
Dan Roberts
Paul Cruz
```

List all of the customers who have placed orders for ACI Widgets (manufacturer ACI, product numbers starting with "4100") between January and June 1990.

```
SELECT COMPANY
FROM CUSTOMERS
WHERE CUST_NUM IN (SELECT DISTINCT CUST
FROM ORDERS
WHERE MFR = 'ACI'
```

```
AND PRODUCT LIKE '4100%'
AND ORDER_DATE BETWEEN '01-JAN-90'
AND '30-JUN-90')
COMPANY
------------------
Acme Mfg.
Ace International
Holm & Landis
JCP Inc.
```

- In each of these examples, the subquery produces a column of data values, and the WHERE clause of the main query checks to see whether a value from a row of the main query matches one of the values in the column. The subquery form of the IN test thus works exactly like the simple IN test, except that the set of values is produced by a subquery instead of being explicitly listed in the statement.

➢ **Existence Test (EXISTS)**

- The existence test (EXISTS) checks whether a subquery produces any rows of query results, as shown in Figure 9-5. There is no simple comparison test that resembles the existence test; it is used only with subqueries.

Figure 9-5: Existence test (EXISTS) syntax diagram

Here is an example of a request that can be expressed naturally using an existence test:
List the products for which an order of $25,000 or more has been received.
The request could easily be rephrased as:
List the products for which there exists at least one order in the ORDERS table (a) that is for the product in question and (b) has an amount of at least $25,000.
The SELECT statement used to retrieve the requested list of products closely resembles the rephrased request:

```
SELECT DISTINCT DESCRIPTION
FROM PRODUCTS
WHERE EXISTS (SELECT ORDER_NUM
FROM ORDERS
WHERE PRODUCT = PRODUCT_ID
AND MFR = MFR_ID
AND AMOUNT >= 25000.00)
DESCRIPTION
------------
500-lb Brace
Left Hinge
Right Hinge
Widget Remover
```

- Conceptually, SQL processes this query by going through the PRODUCTS table and performing the subquery for each product. The subquery produces a column containing the order numbers of any orders for the "current" product that are over $25,000. If there are any such orders (that is, if the column is not empty), the EXISTS test is TRUE. If the subquery produces no rows, the EXISTS test is FALSE. The EXISTS test cannot produce a NULL value.
- You can reverse the logic of the EXISTS test using the NOT EXISTS form. In this case, the test is TRUE if the subquery produces no rows, and FALSE otherwise.
  Notice that the EXISTS search condition doesn't really *use* the results of the subquery at all. It merely tests to see whether the subquery produces any results. For this reason, SQL relaxes the rule that "subqueries must return a single column of data" and allows you to use the SELECT * form in the subquery of an EXISTS test. The previous subquery could thus have been written:

List the products for which an order of $25,000 or more has been received.

```
SELECT DESCRIPTION
FROM PRODUCTS
```

```
WHERE EXISTS (SELECT *
FROM ORDERS
WHERE PRODUCT = PRODUCT_ID
AND MFR = MFR_ID
AND AMOUNT >= 25000.00)
```

In practice, the subquery in an EXISTS test is *always* written using the SELECT * notation.
Here are some additional examples of queries that use EXISTS:
List any customers assigned to Sue Smith who have not placed an order for over $3,000.

```
SELECT COMPANY
FROM CUSTOMERS
WHERE CUST_REP = (SELECT EMPL_NUM
FROM SALESREPS
WHERE NAME = 'Sue Smith')
AND NOT EXISTS (SELECT *
FROM ORDERS
WHERE CUST = CUST_NUM
AND AMOUNT > 3000.00)
COMPANY
--------------
Carter & Sons
Fred Lewis Corp.
```

List the offices where there is a salesperson whose quota represents more than 55 percent of the office's target.

```
SELECT CITY
FROM OFFICES
WHERE EXISTS (SELECT *
FROM SALESREPS
WHERE REP_OFFICE = OFFICE
AND QUOTA > (.55 * TARGET))
CITY
-------
Denver
Atlanta
```

Note that in each of these examples, the subquery includes an outer reference to a column of the table in the main query. In practice, the subquery in an EXISTS test will always contain an outer reference that "links" the subquery to the row currently being tested by the main query.

Q. Write a short note on sub queries restrictions?

- **Subquery restrictions**

A subquery is subject to the following restrictions:

- The *subquery_select_list* can consist of only one column name, except in the **exists** subquery, where an **(*)** is usually used in place of the single column name. Do not specify more than one column name. Qualify column names with table or view names if there is ambiguity about the table or view to which they belong.
- Subqueries can be nested inside the **where** or **having** clause of an outer **select**, **insert**, **update**, or **delete** statement, inside another subquery, or in a select list. Alternatively, you can write many statements that contain subqueries as joins; Adaptive Server processes such statements as joins.
- In Transact-SQL, a subquery can appear almost anywhere an expression can be used, if it returns a single value.
- A subquery can appear almost anywhere an expression can be used. SQL derived tables can therefore be used in the **from** clause of a subquery wherever the subquery is used.

- You cannot use subqueries in an **order by**, **group by**, or **compute by** list.
- You cannot include a **for browse** clause in a subquery.
- You cannot include a **union** clause in a subquery unless it is part of a derived table expression within the subquery. For more information on using SQL derived tables.
- The select list of an inner subquery introduced with a comparison operator can include only one expression or column name, and the subquery must return a single value. The column you name in the **where** clause of the outer statement must be join-compatible with the column you name in the subquery select list.
- *You cannot include text*, *unitext*, or *image* datatypes in subqueries.
- Subqueries cannot manipulate their results internally, that is, a subquery cannot include the **order by** clause, the **compute** clause, or the **into** keyword.
- Correlated (repeating) subqueries are not allowed in the **select** clause of an updatable cursor defined by **declare cursor**.
- There is a limit of 50 nesting levels.
- The maximum number of subqueries on each side of a union is 50.
- The **where** clause of a subquery can contain an aggregate function only if the subquery is in a **having** clause of an outer query and the aggregate value is a column from a table in the **from** clause of the outer query.
- The result expression from a subquery is subject to the same limits as for any expression. The maximum length of an expression is 16K.

Q. Explain Nested sub queries?

➢ **Nested Subqueries**

- All of the queries described thus far in this chapter have been "two-level" queries, involving a main query and a subquery. Just as you can use a subquery "inside" a main query, you can use a subquery inside another subquery. Here is an example of a request that is naturally represented as a three-level query, with a main query, a subquery, and a subsubquery:

List the customers whose salespeople are assigned to offices in the Eastern sales region.

```
SELECT COMPANY
FROM CUSTOMERS
WHERE CUST_REP IN (SELECT EMPL_NUM
FROM SALESREPS
WHERE REP_OFFICE IN (SELECT OFFICE
FROM OFFICES
WHERE REGION =
'Eastern'))
COMPANY
---------------
First Corp.
Smithson Corp.
AAA Investments
JCP Inc.
Chen Associates
QMA Assoc.
Ian & Schmidt
Acme Mfg.
```

In this example, the innermost subquery:

```
SELECT OFFICE
FROM OFFICES
WHERE REGION = 'Eastern'
```

produces a column containing the office numbers of the offices in the Eastern region. The next subquery:

```
SELECT EMPL_NUM
```

```
FROM SALESREPS
WHERE REP_OFFICE IN (subquery)
```

produces a column containing the employee numbers of the salespeople who work in one of the selected offices. Finally, the outermost query:

```
SELECT COMPANY
FROM CUSTOMERS
WHERE CUST_REP IN (subquery)
```

finds the customers whose salespeople have one of the selected employee numbers.

- The same technique used in this three-level query can be used to build queries with four or more levels. The ANSI/ISO SQL standard does not specify a maximum number of nesting levels, but in practice a query becomes much more time-consuming as the number of levels increases. The query also becomes more difficult to read, understand, and maintain when it involves more than one or two levels of subqueries. Many SQL implementations restrict the number of subquery levels to a relatively small number.

Q. Explain correlated sub queries?

➢ **Correlated Subqueries**

In concept, SQL performs a subquery over and over again—once for each row of the main query. For many subqueries, however, the subquery produces the *same* results for every row or row group. Here is an example:

List the sales offices whose sales are below the average target.

```
SELECT CITY
FROM OFFICES
WHERE SALES < (SELECT AVG(TARGET)
FROM OFFICES)
CITY
-------
Denver
Atlanta
```

In this query, it would be silly to perform the subquery five times (once for each office). The average target doesn't change with each office; it's completely independent of the office currently being tested. As a result, SQL can handle the query by first performing the subquery, yielding the average target ($550,000), and then converting the main query into:

```
SELECT CITY
FROM OFFICES
WHERE SALES < 550000.00
```

Commercial SQL implementations automatically detect this situation and use this shortcut whenever possible to reduce the amount of processing required by a subquery. However, the shortcut cannot be used if the subquery contains an outer reference, as in this example:

List all of the offices whose targets exceed the sum of the quotas of the salespeople who work in them:

```
SELECT CITY
FROM OFFICES
WHERE TARGET > (SELECT SUM(QUOTA)
FROM SALESREPS
WHERE REP_OFFICE = OFFICE)
CITY
-----------
Chicago
Los Angeles
```

For each row of the OFFICES table to be tested by the WHERE clause of the main query, the OFFICE column (which appears in the subquery as an outer reference) has a different value. Thus SQL has no choice but to carry out this subquery five times—once for each row in the OFFICES table. A subquery containing an outer reference is called a *correlated subquery* because its results are correlated

with each individual row of the main query. For the same reason, an outer reference is sometimes called a *correlated reference.*

A subquery can contain an outer reference to a table in the FROM clause of any query that contains the subquery, no matter how deeply the subqueries are nested. A column name in a fourth-level subquery, for example, may refer to one of the tables named in the FROM clause of the main query, or to a table named in the FROM clause of the second-level subquery or the third-level subquery that contains it. Regardless of the level of nesting, an outer reference always takes on the value of the column in the "current" row of the table being tested.

Because a subquery can contain outer references, there is even more potential for ambiguous column names in a subquery than in a main query. When an unqualified column name appears within a subquery, SQL must determine whether it refers to a table in the subquery's own FROM clause, or to a FROM clause in a query containing the subquery. To minimize the possibility of confusion, SQL always interprets a column reference in a subquery *using the nearest FROM clause possible*. To illustrate this point, in this example the same table is used in the query and in the subquery:

List the salespeople who are over 40 and who manage a salesperson over quota.

```
SELECT NAME
FROM SALESREPS
WHERE AGE > 40
AND EMPL_NUM IN (SELECT MANAGER
FROM SALESREPS
WHERE SALES > QUOTA)
NAME
-----------
Sam Clark
Larry Fitch
```

The MANAGER, QUOTA, and SALES columns in the subquery are references to the SALESREPS table in the subquery's own FROM clause; SQL does *not* interpret them as outer references, and the subquery is not a correlated subquery. As discussed earlier, SQL can perform the subquery first in this case, finding the salespeople who are over quota and generating a list of the employee numbers of their managers. SQL can then turn its attention to the main query, selecting managers whose employee numbers appear in the generated list.

If you want to use an outer reference within a subquery like the one in the previous example, you must use a table alias to force the outer reference. This request, which adds one more qualifying condition to the previous one, shows how:

List the managers who are over 40 and who manage a salesperson who is over quota and who does not work in the same sales office as the manager.

```
SELECT NAME
FROM SALESREPS MGRS
WHERE AGE > 40
AND MGRS.EMPL_NUM IN (SELECT MANAGER
FROM SALESREPS EMPS
WHERE EMPS.QUOTA > EMPS.SALES
AND EMPS.REP_OFFICE <>
MGRS.REP_OFFICE)
NAME
-----------
Sam Clark
Larry Fitch
```

The copy of the SALESREPS table used in the main query now has the tag MGRS, and the copy in the subquery has the tag EMPS. The subquery contains one additional search condition, requiring that the employee's office number does not match that of the manager. The qualified column name MGRS.OFFICE in the subquery is an outer reference, and this subquery is a correlated subquery.

Q. What do you mean by queries with modified comparison operations?

- **Quantified Tests (ANY and ALL) ***

- The subquery version of the IN test checks whether a data value is equal to some value in a column of subquery results. SQL provides two *quantified tests,* ANY and ALL, that extend this notion to other comparison operators, such as greater than (>) and less than (<). Both of these tests compare a data value to the column of data values produced by a subquery, as shown in Figure 9-6.

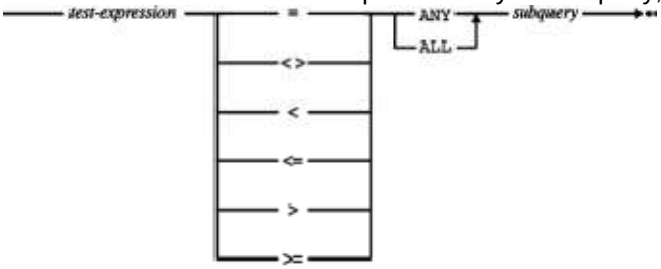


Figure 9-6: Quantified comparison tests (ANY and ALL) syntax diagram

The ANY Test *

- The ANY test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value to a column of data values produced by a subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to *each* data value in the column, one at a time. If *any* of the individual comparisons yield a TRUE result, the ANY test returns a TRUE result.

Here is an example of a request that can be handled with the ANY test:

List the salespeople who have taken an order that represents more than 10% of their quota.

```
SELECT NAME
FROM SALESREPS
WHERE (.1 * QUOTA) < ANY (SELECT AMOUNT
FROM ORDERS
WHERE REP = EMPL_NUM)
NAME
-----------
Sam Clark
Larry Fitch
Nancy Angelli
```

Conceptually, the main query tests each row of the SALESREPS table, one-by-one. The subquery finds all of the orders taken by the "current" salesperson and returns a column containing the order amounts for those orders. The WHERE clause of the main query then computes 10 percent of the current salesperson's quota and uses it as a test value, comparing it to every order amount produced by the subquery. If there is *any* order amount that exceeds the calculated test value, the "< ANY" test returns TRUE, and the salesperson is included in the query results. If not, the salesperson is not included in the query results. The keyword SOME is an alternative for ANY specified by the ANSI/ISO SQL standard. Either keyword can generally be used, but some DBMS brands do not support SOME.

The ANY test can sometimes be difficult to understand because it involves an entire set of comparisons, not just one. It helps if you read the test in a slightly different way than it appears in the statement. If this ANY test appears:

```
WHERE X < ANY (SELECT Y …)
```

instead of reading the test like this:

"where X is less than any select Y…"

try reading it like this:

"where, for *some* Y, X is less than Y"

When you use this trick, the preceding query becomes:

Select the salespeople where, for some order taken by the salesperson, 10% of the salesperson's quota is less than the order amount.

If the subquery in an ANY test produces no rows of query results, or if the query results include NULL values, the operation of the ANY test may vary from one DBMS to another.

The ANSI/ISO SQL standard specifies these detailed rules describing the results of the ANY test when the test value is compared to the column of subquery results:

• If the subquery produces an empty column of query results, the ANY test returns FALSE—there is no value produced by the subquery for which the comparison test holds.

• If the comparison test is TRUE for *at least one* of the data values in the column, then the ANY search condition returns TRUE—there is indeed some value produced by the subquery for which the comparison test holds.

• If the comparison test is FALSE for every data value in the column, then the ANY search condition returns FALSE. In this case, you can conclusively state that there is no value produced by the subquery for which the comparison test holds.

• If the comparison test is not TRUE for any data value in the column, but it is NULL (unknown) for one or more of the data values, then the ANY search condition returns NULL. In this situation, you cannot conclusively state whether there is a value produced by the subquery for which the comparison test holds; there may be or there may not be, depending on the "correct" values for the NULL (unknown) data.

The ANY comparison operator can be very tricky to use in practice, especially in conjunction with the inequality (<>) comparison operator. Here is an example that shows the problem:

List the names and ages of all the people in the sales force who do not manage an office.

It's tempting to express this query as shown on the following page.

```
SELECT NAME, AGE
FROM SALESREPS
WHERE EMPL_NUM <> ANY (SELECT MGR
FROM OFFICES)
```

The subquery:

```
SELECT MGR
FROM OFFICES
```

obviously produces the employee numbers of the managers, and therefore the query *seems* to be saying:

Find each salesperson who is not the manager of any office.

But that's *not* what the query says! What it *does* say is this:

Find each salesperson who, for some office, *is not the manager of that office*.

Of course for any given salesperson, it's possible to find *some* office where that salesperson is not the manager. The query results would include *all* the salespeople and therefore fail to answer the question that was posed! The correct query is:

```
SELECT NAME, AGE
FROM SALESREPS
WHERE NOT (EMPL_NUM = ANY (SELECT MGR
FROM OFFICES))
NAME AGE
-------------- ---
Mary Jones 31
Sue Smith 48
Dan Roberts 45
Tom Snyder 41
Paul Cruz 29
Nancy Angelli 49
```

You can always turn a query with an ANY test into a query with an EXISTS test by moving the comparison *inside* the search condition of the subquery. This is usually a very good idea because it eliminates errors like the one just described. Here is an alternative form of the query, using the EXISTS test:

```
SELECT NAME, AGE
FROM SALESREPS
WHERE NOT EXISTS (SELECT *
FROM OFFICES
WHERE EMPL_NUM = MGR)
```

NAME AGE
---------- ---
Mary Jones 31
Sue Smith 48
Dan Roberts 45
Tom Snyder 41
Paul Cruz 29
Nancy Angelli 49

- **The ALL Test ***

Like the ANY test, the ALL test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value to a column of data values produced by a subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to *each* data value in the column, one at a time. If *all* of the individual comparisons yield a TRUE result, the ALL test returns a TRUE result. Here is an example of a request that can be handled with the ALL test:

List the offices and their targets where all of the salespeople have sales that exceed 50% of the office's target.

```
SELECT CITY, TARGET
FROM OFFICES
WHERE (.50 * TARGET) < ALL (SELECT SALES
FROM SALESREPS
WHERE REP_OFFICE = OFFICE)
```

CITY TARGET
---------- ----------
Denver $300,000.00
New York $575,000.00
Atlanta $350,000.00

Conceptually, the main query tests each row of the OFFICES table, one-by-one. The subquery finds all of the salespeople who work in the "current" office and returns a column containing the sales for each salesperson. The WHERE clause of the main query then computes 50 percent of the office's target and uses it as a test value, comparing it to every sales value produced by the subquery. If *all* of the sales values exceed the calculated test value, the "< ALL" test returns a TRUE, and the office is included in the query results. If not, the office is not included in the query results.

Like the ANY test, the ALL test can be difficult to understand because it involves an entire set of comparisons, not just one. Again, it helps if you read the test in a slightly different way than it appears in the statement. If this ALL test appears:

```
WHERE X < ALL (SELECT Y …)
```

instead of reading like this:

"where X is less than all select Y…"

try reading the test like this:

"where, for *all* Y, X is less than Y"

When you use this trick, the preceding query becomes:

Select the offices where, for all salespeople who work in the office, 50% of the office's target is less than the salesperson's sales.

If the subquery in an ALL test produces no rows of query results, or if the query results include NULL values, the operation of the ALL test may vary from one DBMS to another. The ANSI/ISO SQL standard specifies these detailed rules describing the results of the ALL test when the test value is compared to the column of subquery results:

• If the subquery produces an empty column of query results, the ALL test returns TRUE. The comparison test *does* hold for every value produced by the subquery; there just aren't any values.

• If the comparison test is TRUE for every data value in the column, then the ALL search condition returns TRUE. Again, the comparison test holds true for every value produced by the subquery.

• If the comparison test is FALSE for any data value in the column, then the ALL search

condition returns FALSE. In this case, you can conclusively state that the comparison test does not hold true for every data value produced by the query.
• If the comparison test is not FALSE for any data value in the column, but it is NULL for one or more of the data values, then the ALL search condition returns NULL. In this situation, you cannot conclusively state whether there is a value produced by the subquery for which the comparison test does not hold true; there may be or there may not be, depending on the "correct" values for the NULL (unknown) data.
The subtle errors that can occur when the ANY test is combined with the inequality (<>) comparison operator also occur with the ALL test. As with the ANY test, the ALL test can always be converted into an equivalent EXISTS test by moving the comparison inside the subquery.

Q. Explain SELECT INTO operation?

The **select into** command lets you create a new table based on the columns specified in the **select** statement's select list and the rows chosen in the **where** clause. The **into** clause is useful for creating test tables, new tables as copies of existing tables, and for making several smaller tables out of one large table. You can use **select into** on a permanent table only if the **select into/bulkcopy/pllsort** database option is set to **on**. A System Administrator can turn on this option using **sp_dboption**. Use **sp_helpdb** to see if this option is on.

Here is what **sp_helpdb** and its results look like when the **select into/bulkcopy/pllsort** database option is set to **on**:

```
sp_helpdb pubs2
name      db_size owner   dbid    created      status
---------  ------- ------  -----   -----------  ------------
pubs      2 MB    sa      5       Jun  5 1997  select into
                          /bulkcopy/pllsort

(1 row affected)

device             size        usage
-----------------  ---------   --------------
master             2 MB        data and log

(1 row affected)
```

sp_helpdb output indicates whether the option is set to **on** or **off**. Only the System Administrator or the Database Owner can set the database options.

If the **select into/bulkcopy/pllsort** database option is **on**, you can use the **select into** clause to build a new permanent table without using a **create table** statement. You can **select into** a temporary table, even if the **select into/bulkcopy/pllsort** option is not **on**.

Because **select into** is a minimally logged operation, use **dump database** to back up your database following a **select into**. You cannot dump the transaction log following a minimally logged operation.

Unlike a view that displays a portion of a table, a table created with **select into** is a separate, independent entity. The new table is based on the columns you specify in the select list, the tables you name in the **from** clause, and the rows you choose in the **where** clause. The name of the new table must be unique in the database and must conform to the rules for identifiers.

A **select** statement with an **into** clause allows you to define a table and put data into it, based on existing definitions and data, without going through the usual data definition process.

The following example shows a **select into** statement and its results. A table called *newtable* is created, using two of the columns in the four-column table *publishers*. Because this statement includes no **where** clause, data from all the rows (but only the two specified columns) of *publishers* is copied into *newtable*.

```
select pub_id, pub_name
into newtable
from publishers
(3 rows affected)
```

"3 rows affected" refers to the three rows inserted into *newtable*. Here's what *newtable* looks like:

```
select *
from newtable
pub_id  pub_name
------  ------------------------------------
0736    New Age Books
0877    Binnet & Hardley
1389    Algodata Infosystems
(3 rows affected)
```

The new table contains the results of the **select** statement. It becomes part of the database, just like its parent table.

You can create a skeleton table with no data by putting a false condition in the **where** clause. For example:

```
select *
into newtable2
from publishers
where 1=2
(0 rows affected)
select *
from newtable2
pub_id    pub_name         city      state
------    --------------    --------  -----
```

(0 rows affected)

No rows are inserted into the new table, because 1 never equals 2.

You can also use **select into** with aggregate functions to create tables with summary data:

```
select type, "Total_amount" = sum(advance)
into #whatspent
from titles
group by type
(6 rows affected)
select * from #whatspent
        type        Total_amount
        ------------ ------------------------
        UNDECIDED                    NULL
        business                25,125.00
        mod_cook                 15,000.00
        popular_comp             15,000.00
```

```
psychology          21,275.00
trad_cook           19,000.00
(6 rows affected)
```

Always supply a name for any column in the **select into** result table that results from an aggregate function or any other expression. Examples are:

- Arithmetic aggregates, for example, *amount* *** 2**
- Concatenation, for example, **lname + fname**
- Functions, for example, **lower(lname)**

Here is an example of using concatenation:

```
select au_id,
   "Full_Name" = au_fname + ' ' + au_lname
into #g_authortemp
from authors
where au_lname like "G%"
(3 rows affected)
select * from #g_authortemp
 au_id       Full_Name
 ----------- -------------------------
 213-46-8915 Marjorie Green
 472-27-2349 Burt Gringlesby
 527-72-3246 Morningstar Greene

(3 rows affected)
```

Because functions allow null values, any column in the table that results from a function other than **convert**() or **isnull**() allows null values.

Checking for errors

select into is a two-step operation. The first step creates the new table and the second step inserts the specified rows into the table.

Because **select into** operations are not logged, they cannot be issued within user-defined transactions and cannot be rolled back.

If a **select into** statement fails after creating a new table, Adaptive Server does *not* automatically drop the table or deallocate its first data page. This means that any rows inserted on the first page before the error occurred remain on the page. Check the value of the @@*error* global variable after a **select into** statement to be sure that no error occurred.

If an error occurs from a **select into** operation, **drop table** to remove the new table, then reissue the **select into** statement.

Using *select into* with IDENTITY columns

This section describes special rules for using the **select into** command with tables containing IDENTITY columns.

Selecting an IDENTITY column into a new table

To select an existing IDENTITY column into a new table, include the column name (or the **syb_identity** keyword) in the **select** statement's *column_list*:

select *column_list*
into *table_name*
from *table_name*

The following example creates a new table, *stores_cal_pay30*, based on columns from the *stores_cal* table:

```
select record_id, stor_id, stor_name
into stores_cal_pay30
from stores_cal
where payterms = "Net 30"
```

Q. Explain UNION operation?

The SQL UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

SQL UNION Syntax

```
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2
```

Note: The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.

SQL UNION ALL Syntax

```
SELECT column_name(s) FROM table_name1
UNION ALL
SELECT column_name(s) FROM table_name2
```

PS: The column names in the result-set of a UNION are always equal to the column names in the first SELECT statement in the UNION.

SQL UNION Example

Look at the following tables:

"Employees_Norway":

E_ID	E_Name
01	Hansen, Ola
02	Svendson, Tove
03	Svendson, Stephen
04	Pettersen, Kari

"Employees_USA":

E_ID	E_Name
01	Turner, Sally
02	Kent, Clark
03	Svendson, Stephen
04	Scott, Stephen

Now we want to list **all the different** employees in Norway and USA.

We use the following SELECT statement:

```
SELECT E_Name FROM Employees_Norway
UNION
SELECT E_Name FROM Employees_USA
```

The result-set will look like this:

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Scott, Stephen

Note: This command cannot be used to list all employees in Norway and USA. In the example above we have two employees with equal names, and only one of them will be listed. The UNION command selects only distinct values.

SQL UNION ALL Example

Now we want to list **all** employees in Norway and USA:

```
SELECT E_Name FROM Employees_Norway
UNION ALL
SELECT E_Name FROM Employees_USA
```

Result

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Svendson, Stephen
Scott, Stephen

Q. Write a short note on Sub queries in the HAVING clause?

Subqueries in the HAVING Clause *

Although subqueries are most often found in the WHERE clause, they can also be used in the HAVING clause of a query. When a subquery appears in the HAVING clause, it works as part of the row group selection performed by the HAVING clause. Consider this query with a subquery:

List the salespeople whose average order size for products manufactured by ACI is higher than overall average order size.

```
SELECT NAME, AVG(AMOUNT)
FROM SALESREPS, ORDERS
WHERE EMPL_NUM = REP
AND MFR = 'ACI'
GROUP BY NAME
HAVING AVG(AMOUNT) > (SELECT AVG(AMOUNT)
FROM ORDERS)
NAME AVG(AMOUNT)
---------- ----------
Sue Smith $15,000.00
```

Tom Snyder $22,500.00

Figure 9-7 shows conceptually how this query works. The subquery calculates the "overall average order size." It is a simple subquery and contains no outer references, so SQL can calculate the average once and then use it repeatedly in the HAVING clause. The main query goes through the ORDERS table, finding all orders for ACI products, and groups them by salesperson. The HAVING clause then checks each row group to see whether the average order size in that group is bigger than the average for all orders, calculated earlier. If so, the row group is retained; if not, the row group is discarded. Finally, the SELECT clause produces one summary row for each group, showing the name of the salesperson and the average order size for each.

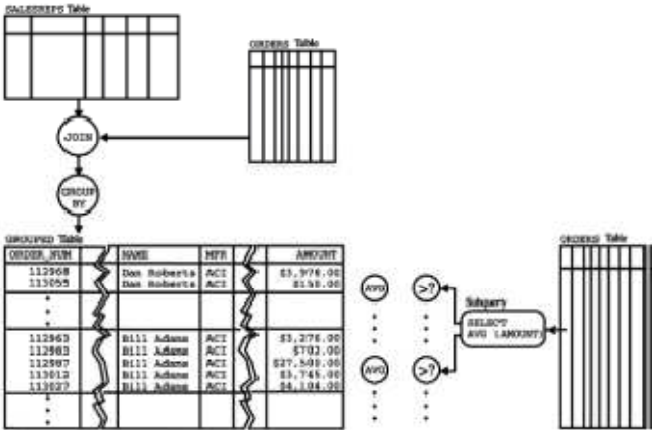


Figure 9-7: Subquery operation in the HAVING clause

You can also use a correlated subquery in the HAVING clause. Because the subquery is evaluated once for each row *group,* however, all outer references in the correlated subquery must be single-valued for each row group. Effectively, this means that the outer reference must either be a reference to a grouping column of the outer query or be contained within a column function. In the latter case, the value of the column function for the row group being tested is calculated as part of the subquery processing.

If the previous request is changed slightly, the subquery in the HAVING clause becomes a correlated subquery:

List the salespeople whose average order size for products manufactured by ACI is at least as big as that salesperson's overall average order size.

```
SELECT NAME, AVG(AMOUNT)
FROM SALESREPS, ORDERS
WHERE EMPL_NUM = REP
AND MFR = 'ACI'
GROUP BY NAME, EMPL_NUM
HAVING AVG(AMOUNT) >= (SELECT AVG(AMOUNT)
FROM ORDERS
WHERE REP = EMPL_NUM)
NAME AVG(AMOUNT)
---------- ----------
Bill Adams $7,865.40
Sue Smith $15,000.00
Tom Snyder $22,500.00
```

In this new example, the subquery must produce "the overall average order size" for the salesperson whose row group is currently being tested by the HAVING clause. The subquery selects orders for that particular salesperson, using the outer reference EMPL_NUM. The outer reference is legal because EMPL_NUM has the same value in all rows of a group produced by the main query

Quick Revision

- A subquery is a "query within a query." Subqueries appear within one of the subquery search conditions in the WHERE or HAVING clause.
- When a subquery appears in the WHERE clause, the results of the subquery are used to select the individual rows that contribute data to the query results.
- When a subquery appears in the HAVING clause, the results of the subquery are used to select the row groups that contribute data to the query results.
- Subqueries can be nested within other subqueries.
- The subquery form of the comparison test uses one of the simple comparison operators to compare a test value to the single value returned by a subquery.
- The subquery form of the set membership test (IN) matches a test value to the set of values returned by a subquery.
- The existence test (EXISTS) checks whether a subquery returns any values.
- The quantified tests (ANY and ALL) use one of the simple comparison operators to compare a test value to all of the values returned by a subquery, checking to see
- whether the comparison holds for some or all of the values.
- A subquery may include an *outer reference* to a table in any of the queries that contain it, linking the subquery to the "current" row of that query.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain Sub queries with IN and EXISTS?	53
2	Q. Write a short note on sub queries restrictions?	55
3	Q. Explain Nested sub queries?	56
4	Q. Explain correlated sub queries?	57
5	Q. What do you mean by queries with modified comparison operations?	59
6	Q. Explain SELECT INTO operation?	62
7	Q. Explain UNION operation?	65
8	Q. Write a short note on Sub queries in the HAVING clause?	67