Chapter 5

SQL

SQL is also a declarative query language, which indicates the properties of the data to be retrieved. The basic query expression in SQL is a syntactic variation of an expression in TRC and is easily mapped to its equivalent relational algebra expression. Since SQL is an industry-standard query language, it must be relationally complete. This text illustrates the relational completeness of SQL by providing the SQL expressions for the same queries that illustrated the fundamental and additional operators of relational algebra. The examples over the EMPLOYEE TRAINING enterprise continue to use basic features of SQL to illustrate the relationship between SQL and the formal query languages. For pragmatic reasons, SQL is more powerful than its formal counterparts on which it is based. These additional features of SQL, including ordering and aggregation of results, are illustrated by example. The SQL standard evolved to provide more than a query language, including a DDL, a DML, and a data control language (DCL) for integrity and security.

The SQL standard, originally adopted by the American National Standards Institute (ANSI) in 1986 and the International Standards Organization (ISO) in 1987, contains many components and continues to evolve, with additional versions ratified in 1989, 1992, and 1999. The latest standard SQL:1999 includes support for object-oriented extensions of the relational model, including user-defined data types and object tables, but is beyond the scope of this book, which focuses only on the pure relational model. The 1989 and 1992 standards, SQL-89 and SQL-92, respectively, extended the original standard with various features, which are too numerous to mention in their entirety here. However, the programmatic interfaces to SQL were primarily defined in these revisions to the standard. Since SQL is not a general-purpose programming language, the standard requires that an

implementation of SQL support at least one method by which SQL can be invoked programmatically, such as a module, embedded SQL, the Call-Level Interface, and direct SQL. Since the focus of this book is on query languages, this chapter does not cover the programmatic interfaces to SQL, but provides an overview of the ad hoc capabilities of SQL. In general, this overview is compatible with the SQL-92 standard, although the chapter coverage chooses to start with basic SQL query expressions and extends the coverage to include the explicit support for joins, which was added to the standard in SQL-92.

5.1 BASIC QUERY EXPRESSIONS

Although there are always slight variations of the dialect of SQL provided by a given system, the SQL standard provides a description of the language that allows for the portability of database applications. Identifiers in the SQL standard are limited to a maximum of 128 characters; the first character of an identifier must be a letter (upper- or lowercase) and the remaining may be any letter, digit, or underscore. Identifiers are case insensitive in SQL, so the identifiers `tableName` and `tablename` are considered identical. Thus, the syntax conventions for identifiers used in this book are not inconsistent with the SQL standard. Specifically, the SQL examples continue to use identifiers starting with a lowercase letter to refer to the name of a relation or attribute.

The basic SQL query expression selects a list of attributes from relations where a condition holds:

```
select a1, a2, ..., aj
from   r1, r2, ..., rk
where  θ
```

This basic SQL query expression is essentially equivalent to the relational algebra expression

$$\pi_{a1,a2,\ldots,aj}(\sigma_\theta(r1 \times r2 \times \ldots \times rk)).$$

The `select` clause *projects* the desired attributes from the cartesian product of the tables that satisfy the *selection* condition specified by the `where` clause. It is interesting to note that the SQL `select` clause is in fact a relational algebra *projection* and not a *selection*. Another point of interest to note is that the `from` clause represents a *cartesian product* of the tables specified. One of the most common mistakes in SQL is not realizing that the `from` clause provides all possible combinations of the tuples from the tables specified in the list. If a relational algebra *join* is desired, then the join condition is specified as part of the `where` clause, which performs a selection on the result of the cartesian product.[1]

[1] SQL-92 introduced a *join* syntax in the `from` clause, which will be described later in this chapter.

There are slight variations on this SQL syntax. If all attributes are desired in the result, an asterisk (*) provides a convenient shorthand:

```
select *
from   r1, r2, ..., rk
where  θ
```

If there is no selection condition, the `where` clause is optional:

```
select *
from   r1, r2, ..., rk
```

The equivalence of the basic SQL expression to its corresponding relational algebra expression was qualified as *essentially equivalent*. The relational algebra and relational calculus languages assume that relations are *sets* of tuples. A set is a collection of elements without duplicates. Therefore, the formal languages implicitly remove duplicate tuples from the result set. However, SQL assumes that a relation is a *multiset*, or *bag*, of tuples, which is a collection of elements that may contain duplicates. Since there is a cost associated with performing duplicate elimination, SQL chooses as a default to keep duplicate tuples. Duplicate elimination is possible using the keyword `distinct` in the select clause:

```
select distinct a1, a2, ..., aj
from   r1, r2, ..., rk
where  θ
```

There are also circumstances that require the use of multisets. Consider the situation where the sum of the salaries of employees is needed to determine the salary budget for a company. If the sum is to be performed on a set of employee salaries, rather than a multiset of salaries, then the result of the sum on the set would be incorrect if there were employees earning the same salary.

This basic SQL query expression is a syntactic variation of the TRC expression:

```
{R1, R2, ..., RK | r1(R1) and r2(R2) and ... rk(Rk) and θ}
```

In the TRC expression, the tuple variables to the left of the vertical bar correspond to the SQL `select` clause, specifying the schema of the resulting relation. The tables in the SQL `from` clause correspond to the table references listed to the right of the vertical bar, which specifies that `Ri` is a tuple variable ranging over the table `ri`. The condition of the `where` clause, denoted θ, is part of the condition of the TRC expression.

In this brief comparison of TRC versus SQL, the differences appear minor. The TRC shortcut for including all attributes appearing in the query expression is to list all of the tuple variables, whereas SQL uses an asterisk. When all attributes corresponding to a tuple variable are not desired, dot notation `T.a` projects the

attribute a associated with the tuple variable T. The foregoing SQL syntax show
a list of attribute names. However, since attribute names are only unique within
table, a reference to an attribute name may be ambiguous. SQL also allows the us
of dot notation to refer to an attribute and requires dot notation when an attribut
reference is ambiguous. For example, if two tables t1 and t2 both contain th
attribute a, then the select clause in the following query contains an ambiguou
reference to the attribute a:

```
select a
from   t1, t2
where  ...
```

However, dot notation t1.a disambiguates the reference by specifying the table
the desired attribute:

```
select t1.a
from   t1, t2
where  ...
```

SQL allows for the specification of *table aliases* for a table in the from claus
There are several reasons for introducing table aliases. One reason is to provide
short name to reference tables, as in the following:

```
select T1.a, T2.b
from   longTableName1 T1, longTableName2 T2
where  ...
```

Another reason is to provide a mechanism by which the same table can be access
more than once in the same query. These table aliases (sometimes called *ran*
variables) play a similar role to the tuple variables in TRC. This similarity will
made obvious in the examples describing the SQL language by following the sam
conventions that were used for describing the TRC language. For example, th
identifiers used to denote the preceding table aliases begin with an uppercase lett
as was done in TRC.

For consistency of presentation of the query languages, the examples illu
trating SQL also assume the same syntax for intermediate tables and renaming
attributes:

```
intermediateTable(attr1, ..., attrn) := queryExpression;
```

If renaming of attributes is not desired, the syntax

```
intermediateTable := queryExpression;
```

derives the attribute names for the schema of intermediateTable from
queryExpression. SQL has its own syntax for defining tables and renam
attributes, which is introduced later in this chapter.

5.2 EXPRESSIVE POWER

SQL is relationally complete, and following the explanation of the other query languages, this section gives a corresponding SQL query expression for the fundamental and additional operators of relational algebra. The focus of this exposition is on the expressive power of the basic SQL expression to emphasize the correlation between SQL and TRC. To make this relationship more obvious, the examples use mnemonic table aliases that are identifiers starting with an uppercase letter similar to that of TRC. Also, a table alias is often used to refer to an attribute, even when it is not necessary to disambiguate the attribute reference.

5.2.1 Fundamental Operators

A relationally complete language must provide at least the equivalence of the fundamental relational algebra operators ($\sigma, \pi, \cup, -, \times$). Table 5.1 summarizes the SQL expressions for abstract operations involving the fundamental relational algebra operators.

TABLE 5.1 SQL summary of fundamental relational algebra operators.

Algebra	SQL
$\sigma_\theta(r)$	`select * from r where` θ
$\pi_A(r)$	`select distinct A from r`
$r \cup s$	`select * from r union select * from s`
$r - s$	`select * from r except select * from s`
$q \times r$	`select * from q, r`

Table 5.2 provides the SQL queries that correspond to the illustrative examples over the EMPLOYEE TRAINING enterprise using the fundamental operators. Query Q_σ is a straightforward selection of an employee having a salary greater than $100,000.

Query Q_π projects the desired attributes of the table alias E that ranges over the employee table. Since the key attribute eID of the employee table is not included in the result of the query, duplicates may occur in the result. Therefore, distinct is added to the select clause to return a *set* of tuples to be equivalent to the relational algebra specification.

Query $Q_\cup$ unions together the SQL query expressions that find employees who are managers and employees who are coaches. Note the SQL queries that are operands of the union query are compatible. In general, the union requires that the tables have the same number of attributes and that the types of the attributes are compatible.

TABLE 5.2 SQL summary of fundamental EMPLOYEE TRAINING queries.

Query	√SQL
Q_σ	`select * from employee E where E.eSalary > 100000;`
Q_π	`select distinct E.eLast, E.eFirst, E.eTitle from employee E;`
$Q_\cup$	`select E.eID from employee E where E.eTitle='Manager' union select E.eID from employee E where E.eTitle='Coach';`
Q_-	`select E.eID from employee E where E.eTitle='Manager' except select T.eID from takes T;`
$Q_\times$	`select E.eID, C.cID from employee E, trainingCourse C ;`

By default, SQL removes duplicate tuples in union queries. To allow duplicates the option `all` can be specified after the `union`. For example,

```
select * from r union all select * from s
```

If a tuple t appears t_r times in `r` and t_s times in `s`, then t appears $t_r + t_s$ times in the resulting union all. The definition of how an operator deals with duplicates, as just described for the `union all`, is known as its *duplicate semantics*.

The previous version of the "Managers or Coaches" query purposely follows the general template for expressing a union in SQL. However, this particular query can also be phrased as the single query:

√
```
select *
from   employee E
where  E.eTitle = 'Manager' or E.eTitle = 'Coach';
```

Query Q_- uses the `except` operator to return the employees who are managers and have not taken a training course. The `except` operator has similar assumptions to that of the `union` operator on type compatibility and duplicate removal. The option `all` is also available on the `except` operator:

```
select * from r except all select * from s
```

The duplicate semantics of the `except all` for a tuple t appearing t_r times in `r` and t_s times in `s` is the maximum of zero or $t_r - t_s$.

Query $Q_\times$ represents a cartesian product of the `employee` and `trainingCourse` tables. To be consistent with the relational algebra specification for the query, the result of the query includes only the `eID` and `cID` attributes of the tables aliases `E` and `C`, respectively. Recall that the simplifying assumption for the cartesian product operator in relational algebra requires that the operand relations not have any attribute names in common, so that the result of the cartesian product does not contain duplicate attribute names. Since the selective renaming of columns in SQL has not yet been introduced, this exposition makes a similar assumption.

5.2.2 Additional Operators

The additional relational algebra operators ($\cap$, $\bowtie_\theta$, $\bowtie$, $\div$) provide an abbreviation for frequently used combinations of the fundamental operators. The SQL queries that correspond to the illustrative examples over the EMPLOYEE TRAINING enterprise using these additional operators of relational algebra are summarized in Table 5.3.

TABLE 5.3 SQL summary of additional EMPLOYEE TRAINING queries.

Query	√SQL
$Q_\cap$	`select E.eID from employee E where E.eTitle='Manager' intersect select T.eID from takes T;`
$Q_{\bowtie_\theta}$	`select * from employee E, technologyArea A where E.eID=A.aLeadID;`
$Q_\bowtie$	`select distinct C.cTitle, T.tYear, T.tMonth, T.tDay from trainingCourse C, takes T where C.cID=T.cID ;`

Query $Q_\cap$ finds the employees who are managers and have taken a training course. The type compatibility and duplicate removal assumptions of the union operator apply to the `intersect` operator, including the `all` option:

```
select * from r intersect all select * from s
```

If a tuple t appears t_r times in `r` and t_s times in `s`, then the duplicate semantics of the `intersect all` results in the tuple t appearing the minimum of t_r or t_s times.

Query $Q_{\bowtie_\theta}$ illustrates a join of the `employee` and `technologyArea` tables such that the employee `eID` is equal to the `aLeadID` of the technology area.

Query $Q_\bowtie$ joins the `trainingCourse` and `takes` tables on the `cID` attributes in the where clause. The `distinct` in the select clause eliminates the many duplicates that otherwise would be included in the result, since the multiset version would include a tuple for each employee taking the course on a given date.

In relational algebra, the natural join ($\bowtie$) provides a convenient shorthand for joining two tables such that the value of attributes with the same name in both operand relations are equal. A projection is automatically introduced to include only one copy of the duplicate attributes. DRC provides a shortcut for natural join by using the same domain variable name in the positions that are to be (natural) joined. Both TRC and the basic SQL query expression do not provide for a natural join shortcut. However, SQL-92 introduced a natural join syntax in the `from` clause, which is described later in this chapter.

The discussion of the relational algebra division operator is deferred to the next section, which introduces exists as part of a conditional expression in SQL, which is essential to the specification of division in SQL.

5.2.3 Exists

The examples thus far illustrate the similarity of the basic SQL query expression to a TRC expression. However, the definition of a TRC formula includes the subformula (exists T)$\mathcal{F}$(T), which is TRUE if there exists a value assigned to T that makes $\mathcal{F}$(T) TRUE; otherwise, the subformula is FALSE. SQL also allows for `exists` as part of its conditional expression in the `where` clause. Specifically, the syntax is

`exists` (*table-expression*)

where `exists` is TRUE when the *table-expression* does not result in an empty table; otherwise, the `exists` is FALSE. In the TRC chapter, the `exists` subformula appeared in the examples describing the difference, intersection, and division operators of relational algebra in the context of relational calculus.

Difference. The difference example query over the EMPLOYEE TRAINING enterprise retrieves the identification number of employees who are managers that have *not* taken any training courses. The previous version of the query used the except operator in SQL after defining the `managers` and `takenCourse` intermediate tables:

```
select * from managers except select * from takenCourse;
```

This query can also be specified in one step in SQL using an `exists` condition in the `where` clause:

✓
```
select E.eID
from   employee E
where  E.eTitle='Manager' and not exists
       (select  *
       from     takes T
       where    T.eID=E.eID);
```

The `select-from-where` table expression appearing in the `where` clause is a nested query, which is nested inside the outer `select-from-where` query. If the nested query returns an empty table, indicating that the employee has not taken any training course, then the `exists` evaluates to FALSE and the `not` evaluates to TRUE. If the conditional `E.eTitle='Manager'` also evaluates to TRUE, then the employee is included in the result of the (outer) query.

This alternative specification of the `except` query is an example of a *nested correlated subquery*. The query `select * from takes T where T.eID=E.eID` is a nested subquery that is correlated by referencing the value of the `eID` attribute of the `employee` table from the outer query within the nested subquery.

Intersection. The intersection example query over the EMPLOYEE TRAINING enterprise retrieves the identification number of employees who are managers and

have taken at least one training course:

```
select * from managers intersect select * from takenCourse;
```

The `intersect` query can also be specified in one step using a similar nested correlated subquery:

✓
```
select E.eID
from   employee E
where  E.eTitle='Manager' and exists
       (select  *
       from     takes T
       where    T.eID=E.eID);
```

In this case, an employee that is a manager is included in the result if there exists a tuple in the `takes` table for that employee.

Division. The division example query over the abstract domain finds the a's from the `abTable` that are related to *all* of the b's specified in the `bTable`, where the schema of the tables are `abTable(a,b)` and `bTable(b)`. Since SQL does not provide for the specification of universal quantification, SQL's specification of division uses the logically equivalent definition in terms of existential quantification:

✓
```
select distinct T.a
from   abTable T
where  not exists
       (select  *                                        -- β
       from     bTable B
       where    not exists
                (select  *                               -- αβ
                from     abTable AB
                where    AB.a=T.a and AB.b=B.b));
```

This division query finds the a's from the `abTable` such that it is not the case that there exists a `b` value in the `bTable` that is *not* related to the `a` value by the `abTable`. Recall that the instance of `bTable` contains the tuples `{ (b1),(b2) }` and that the instance of `abTable` contains the tuples `{(a1,b1), (a1,b2), (a2,b2), (a3,b1), (a3,b2), (a3,b3)}`. Therefore, the answer to the query is `{(a1),(a3)}`.

Table 5.4 shows a truth table that corresponds to the specification of division in SQL on the abstract example. There are two nested queries in this example, which are labelled by comments in the SQL specification. The innermost nested query $\alpha\beta$ checks whether the `a` value from the outermost query (`T.a`) is related by the `abTable` to the `b` value from the inner query (`B.b`). The nested query β checks whether there exists a `b` value in the `bTable` that is not related by the

abTable to the a value specified in the outermost query (T.a). The truth table indicates that the a1 and a3 values from the abTable are related to *all* of the values in the bTable (i.e., related to b1 and b2). The value a2 is not included in the result, since there exists a b value, specifically b2, that is not related to a2 by the abTable.

TABLE 5.4 SQL division truth table.

T.a	B.b	exists $\alpha\beta$	not exists $\alpha\beta$	exists β	not exists β
a1	b1	T	F	F	T
	b2	T	F		
a2	b1	T	F	T	F
	b2	F	T		
a3	b1	T	F	F	T
	b2	T	F		

5.2.4 Safety

The previous examples have demonstrated the relational completeness of SQL illustrating how the operators of relational algebra can be expressed in SQL. Recall that a *safe* expression guarantees a finite result, and it is well known that relational algebra and safe TRC (and safe DRC) are equivalent in expressive power. Is SQL safe? Yes. The syntax of SQL guarantees the safety of the SQL language. Since an exists or not exists condition must appear only in a where clause that is associated with a (select and) from clause, a basic SQL query expression limits a table alias to a positive table reference in the from clause. The example queries over the EMPLOYEE TRAINING enterprise used in the safety discussion for relational calculus languages finds the employees that do not lead technology areas, assuming that there exists an intermediate table leads containing the IDs of the employees that lead technology areas:

✓
```
select E.eID
from   employee E
where  not exists
       (select *
       from    leads L
       where   L.eID=E.eID);
```

The table alias E is inherently limited to tuples in the employee table and the eID attribute is checked in the nested subquery to determine whether the employee leads a technology area.

5.3 EXAMPLE QUERIES

This section illustrates the six EMPLOYEE TRAINING example queries using the basic SQL query expression, continuing to emphasize its similarity to TRC. For example, queries Q4 and Q5 continue to use the creative relational calculus approach to answer a more than one and minimum query, even though SQL provides inherent support for counting and finding the minimum or maximum. The next section explores the additional capabilities of SQL. In addition to introducing new queries to illustrate these features, we present alternative solutions to queries Q1 through Q5. The approach to answer the division query Q6 is the same in both SQL and TRC: finding an employee who has taken at least one database course and it's not the case that there exists a database course that the employee has not taken.

✓Q1: What training courses are offered in the 'Database' technology area?

```
(cID, cTitle, cHours)

dbCourse :=
  select C.cID, C.cTitle, C.cHours
  from   trainingCourse C
  where  exists
         (select *
         from    technologyArea A
         where   A.aID = C.areaID and
                 A.aTitle = 'Database');
```

✓Q2: Which employees have taken a training course offered in the 'Database' technology area?

```
(eID, eLast, eFirst, eTitle)

dbEmployee :=
        select  E.eID, E.eLast, E.eFirst, E.eTitle
        from    employee E
        where   exists
                (select *
                from    takes T, dbCourse D
                where   T.eID=E.eID and T.cID=D.cID);
```

√Q3: Which employees have not taken any training courses?

```
(eID, eLast, eFirst, eTitle)

select  E.eID, E.eLast, E.eFirst, E.eTitle
from    employee E
where   not exists
        (select *
        from    takes T
        where   T.eID=E.eID);
```

√Q4: Which employees took courses in more than one technology area?

```
(eID, eLast, eFirst, eTitle)

select  E.eID, E.eLast, E.eFirst, E.eTitle
from    employee E
where   exists
        (select *
        from    takes T1, takes T2,
                trainingCourse C1, trainingCourse C2
        where   T1.eID=E.eID and T2.eID=E.eID and
                T1.cID=C1.cID and T2.cID=C2.cID and
                C1.areaID <> C2.areaID);
```

√Q5: Which employees have the minimum salary?

```
(eID, eLast, eFirst, eTitle, eSalary)

select  E.eID, E.eLast, E.eFirst, E.eTitle, E.eSalary
from    employee E
where   not exists
        (select *
        from    employee S
        where   S.eSalary < E.eSalary);
```


√Q6: Which employees took all of the training courses offered in the 'Database' technology area?

```
(eID, eLast, eFirst, eTitle)

select  E.eID, E.eLast, E.eFirst, E.eTitle
from    employee E
where   exists
                (select *
                from    dbEmployee B
                where   B.eID=E.eID) and
        not exists
                (select *
                from    dbCourse D
                where   not exists
                        (select *
                        from    takes T
                        where   T.eID=E.eID and
                                T.cID=D.cID));
```

5.4 ADDITIONAL FEATURES OF THE QUERY LANGUAGE

Up to this point, this chapter has emphasized the similarity between the basic SQL query expression and the formal TRC language on which it is based. Besides the syntactic variation, the only difference pointed out so far is that the formal languages assume that relations are sets of tuples, whereas SQL assumes that relations are multisets of tuples, which allows duplicates. Since SQL is an industry-standard query language, it was designed to be more powerful than TRC. These additional features of SQL include views or virtual tables, sorting and aggregation of results, nested subqueries involving equality and set comparisons, and the flexibility added to the query language in SQL-92 for using joined tables in the `from` clause.

5.4.1 Views and Renaming of Attributes

Throughout the chapters of this text, intermediate tables with optional renaming of attributes provide a mechanism to break down a query into logical pieces. A similar level of abstraction is built into the SQL language as *views* or *virtual tables*. For example, the view `leadsArea` gives the identification number of employees with the technology area that the employee leads:

```
create  view leadsArea as
select  aLeadID, aID
from    technologyArea
```

Both of the definitions of intermediate tables and virtual tables relate a table expression to a table name. However, there is a fundamental difference between intermediate tables and views. When an intermediate table is defined, its defining expression is executed, and the resulting tuples are stored. If the tables appearing in the defining expression change after the intermediate table is defined, the intermediate table will not reflect these changes. In contrast, when a view is defined, its defining expression is stored and executed each time the view is referenced. Therefore, the view will always reflect any changes in the tables appearing in the defining expression. There is, however, a trade-off between the recomputation of a view on each reference versus the inability of an intermediate table to reflect changes from tables in its defining expression.

SQL also supports a concept similar to an intermediate table by its insert data manipulation statement, which allows for the specification of a table expression as the source of the insertion for a named table. A detailed example is provided in the section covering SQL's data manipulation language.

The `leadsArea` virtual table derives the names of the attributes from its defining expression. However, the `create view` statement supports an optional commalist for the renaming of attributes. For example,

```
create  view leadsArea(leadID, areaID) as
select  aLeadID, aID
from    technologyArea
```

renames the `aLeadID` attribute to `leadID` and the `aID` attribute to `areaID`.

Similar to the earlier syntax for renaming attributes while defining an intermediate table, the optional commalist specifies all attributes in the table's schema. However, SQL also supports selective column renaming in the `select` clause.

```
create  view leadsArea as
select  aLeadID, aID as areaID
from    technologyArea
```

In this example, the schema of the view is `leadsArea(aLeadID, areaID)` where the name of the first attribute `aLeadID` is derived from the expression and the name of the second column is explicitly renamed to `areaID`. This capability of SQL for selective column renaming in the `select` clause is quite useful. Additional examples of the use of selective column renaming in queries that aggregate results appear later in this chapter.

5.4.2 Sorting

The query languages assume that relations are sets or multisets of tuples, which implies an unordered collection of tuples. However, it is quite useful in practice to display the results of a query in a particular order. SQL provides an `order by` clause

for that purpose. For example, the following query displays employee information ordered by the employee's last name:

```
✓ 01:  select     E.eLast, E.eTitle, E.eSalary
       from       employee E
       order by   E.eLast;
```

The `order by` clause also allows for the ordering of multiple attributes and for the specification of the sort order as ascending or descending. In the previous query, the sort order is assumed to be ascending, which is the default sort specification. The next query displays the titles of employees in ascending (alphabetical) order and within that displays the salaries of the employees with the same title in descending order:

```
02:    select     E.eTitle, E.eSalary
       from       employee E
       order by   E.eTitle asc, E.eSalary desc;
```

The specification of the attributes in the `order by` clause may use a position reference instead of an attribute name:

```
03:    select     E.eTitle, E.eSalary
       from       employee E
       order by   1 asc, 2 desc;
```

Although SQL allows this by-position syntax specification for the `order by` clause, the by-name syntax is more readable. Since SQL allows for the selective renaming of a column in the select clause, the by-name syntax is preferred.

As indicated, the `order by` clause sorts the tuples in the result of a query. The `order by` clause cannot be used in an arbitrary table expression. Tables in SQL are still unordered collections of tuples. Therefore, the `order by` clause can not be used in a view definition since views are virtual tables.

5.4.3 Aggregation

Many important queries over the data in a database involve aggregation of results, such as queries involving minimum, maximum, sum, count, and average. The formal query languages of relational algebra and relational calculus do not provide inherent support for aggregation, although queries involving minimum or maximum and a limited form of counting (more than one or only one) can be expressed in these languages, as illustrated in the earlier chapters. However, SQL provides inherent support in the language for aggregation constructs.

Consider the following query that finds the minimum, maximum and average employee salary, as well as the sum of all salaries and a count of the number of

employees in the database:

```
√A1:  select  min(E.eSalary), max(E.eSalary),
              avg(E.eSalary), sum(E.eSalary), count(*)
      from    employee E;
```

This query returns a single tuple with five attribute values. The `min`, `max`, `avg` and `sum` aggregate functions apply to a numeric attribute. The `count(*)` counts the number of tuples in the result of the `from` clause, since there is no `where` clause specified.

As another example, consider a query that counts the number of employees that took a training course in the database technology area:

```
√A2:  select  count(distinct T.eID)
      from    dbCourse D, takes T
      where   D.cID = T.cID;
```

Since an employee can take many training courses in the database technology area, the count aggregate function counts the number of `distinct` employee IDs.

Grouping. In the previous example, the aggregation applied to all of the tuples in the result of the (`from` and) `where` clause. Some queries require applying an aggregate function to groups of tuples that have the same value on a grouping of attributes. SQL provides a `group by` clause for the specification of the grouping attributes. For example, consider the query that finds the minimum, maximum, and average salary of employees by title:

```
√G1:  select    E.eTitle, min(E.eSalary),
                max(E.eSalary), avg(E.eSalary)
      from      employee E
      group by  E.eTitle;
```

The database system groups together the tuples that have the same value of `eTitle` and finds the minimum, maximum, and average `eSalary` of the employees in each group.

As another example, for each technology area, find the number of employees that took training courses in that technology area. Display the identification number of the technology area, its title, and the number of employees:

```
√G2:  select    A.aID, A.aTitle,
                count(distinct T.eID) as numEmps
      from      technologyArea A, trainingCourse C, takes T
      where     A.aID = C.areaID and
                C.cID = T.cID
      group by  A.aID, A.aTitle;
```


Notice that the `group by` clause specifies both the `aID` and `aTitle` attributes. If only the `aID` attribute is specified in the `group by` clause while both the `aID` and `aTitle` attributes are specified in the `select` clause, SQL does not know what to do if more than one value of `aTitle` occurs in combination with an `aID` value. Therefore, SQL requires that any nonaggregate columns appearing in the `select` clause appear in the `group by` clause, so that the aggregates produce a single result for each group.

Having. SQL also supports the ability to place a selection condition on the results of grouping by the `having` clause. For example, consider a modification of the query G2 that identifies technology areas with at least 4 employees that took training courses in that technology area and displays the result in descending order on the number of employees:

```
H1:   select    A.aID, A.aTitle,
                count(distinct T.eID) as numEmps
      from      technologyArea A, trainingCourse C, takes T
      where     A.aID = C.areaID and
                C.cID = T.cID
      group by  A.aID, A.aTitle
      having    numEmps >= 4
      order by  numEmps desc;
```

The `having` clause avoids a two-step process. Assume that the query G2 is the defining expression for a view named `numEmpsTakenArea`; then the following query generates the same results as query H1:

```
√H2:  select    *
      from      numEmpsTakenArea
      where     numEmps >= 4
      order by  numEmps desc;
```

The query Q4 that finds the employees who took courses in more than one technology area can also be rephrased using aggregation, grouping, and having. Query Q4A*i* defines a view `moreThanOneTechArea` that finds the employees who took courses in more than one technology area:

```
Q4Ai: create    view moreThanOneTechArea as
      select    T.eID, count(distinct areaID) as numAreas
      from      takes T, trainingCourse C
      where     T.cID = C.cID
      group by  T.eID
      having    numAreas > 1;
```

Query Q4A*ii* joins the view defined in Q4A*i* with the employee table to return the desired attributes:

```
Q4Aii:  select  E.eID, E.eLast, E.eFirst, E.eTitle
        from    employee E
        where   exists
                (select *
                from    moreThanOneTechArea M
                where   M.eID=E.eID);
```

5.4.4 Nested Subqueries

Earlier in the chapter, the exists and not exists subqueries illustrated nested subqueries and the use of existential conditions in the where clause to support division and to optionally support the specification of difference and intersection. Nested subqueries also provide a mechanism to support equality and set comparisons.

Equality Comparison. Consider another specification of Q1 that finds the courses in the database technology area:

```
√Q1A:   create view dbCourse as
        select C.cID, C.cTitle, C.cHours
        from   trainingCourse C
        where  C.areaID =
               (select   A.aID
               from      technologyArea A
               where     A.aTitle = 'Database');
```

The conditional expression in the where clause allows for an equality comparison between the areaID attribute of the trainingCourse table and the result of a select-from-where expression that returns the identification number of the database technology area.

There are constraints on the use of the equality comparison. The nested subquery must return a single tuple, and the items being compared must be compatible. In Q1A, the nested subquery retrieves the key of the technologyArea table, and it is compared to its referencing foreign key in the trainingCourse table.

The nested subquery of Q1A is not a correlated subquery. The nested subquery is executed once to find the identification number of the database technology area, which is then compared with the areaID attribute of a trainingCourse. The earlier existential examples presented were nested correlated subqueries where the table aliases in the outer query are referenced in the subquery. Thus, the correlated subquery is evaluated for each value in the outer query.

Besides being used to compare a foreign key with the primary key that it references, equality comparisons are useful in the presence of aggregation. Query Q5 retrieved the employee information for those employees having the minimum salary: finding those employees where it is not the case that there exists another employee that has a smaller salary.

Consider another specification of this query using the inherent aggregation capabilities of SQL to find the employees whose salary is equal to the minimum employee salary.

```
√Q5A:   select  E.eID, E.eLast, E.eFirst, E.eTitle
        from    employee E
        where   E.eSalary =
                (select min(S.eSalary)
                from    employee S );
```

The nested subquery of Q5A is not correlated, since it finds the minimum salary once, which is compared with an employee's salary in the conditional expression of the where clause.

Set Comparison. The conditional expression in the where clause also allows for the specification of comparisons based on set membership using the in and not in operators. Some of the example nested correlated subqueries can be rewritten to noncorrelated nested subqueries using set comparisons.

For example, query Q2 finds those employees who have taken a training course in the database technology area. Query Q2A answers the same query by retrieving the requested employee information for those employees whose eID value appears in the multiset of employee IDs who have taken a course in the database technology area:

```
√Q2A:   select  E.eID, E.eLast, E.eFirst, E.eTitle
        from    employee E
        where   E.eID in
                (select T.eID
                from    takes T, dbCourse D
                where   T.cID=D.cID);
```

As another example, query Q3 finds those employees that have not taken any training courses. Query Q3A answers the same query by retrieving the requested employee information for those employees whose eID value does *not* appear in the multiset of employee IDs that have taken any training course:

```
√Q3A:   select  E.eID, E.eLast, E.eFirst, E.eTitle
        from    employee E
        where   E.eID not in
```

```
        (select T.eID
        from    takes T);
```

Both of the preceding examples replace a nested correlated subquery using an existential comparison with a nested uncorrelated subquery using a set comparison. The nested correlated subquery is executed once for each value in the outer query, whereas the nested uncorrelated subquery is executed once, and each value in the outer query is checked for membership in the resulting multiset.

5.4.5 Joined Tables

Recall the query Q1 that finds the training courses offered in the database technology area. The original specification of Q1 followed the TRC specification of the query using an existential condition in the `where` clause. The query Q1A specified the same query using the `in` set comparison operator in the `where` clause. The query can also be specified using join conditions in the `where` clause. In query Q1B, the `trainingCourse` and the `technologyArea` tables are joined by the condition `C.areaID = A.aID` in the `where` clause:

```
√Q1B:   create view dbCourse as
        select C.cID, C.cTitle, C.cHours
        from   trainingCourse C, technologyArea A
        where  C.areaID = A.aID and A.aTitle = 'Database';
```

One of the most common mistakes in forming a query expression in SQL is omitting of the join conditions in the `where` clause of the query. In the foregoing example, if the join condition (`C.areaID = A.aID`) is omitted, then the result contains the database technology area combined in a cartesian product with all of the rows in the `trainingCourse` table.

The SQL-92 revision of the standard introduced *joined tables* to allow the specification of a join in the `from` clause of an SQL query. In the Q1C specification of the query, the `trainingCourse` and `technologyArea` tables are explicitly joined in the `from` clause on their related attributes rather than the implicit join of Q1 specified in the `where` clause:

```
Q1C:    create view dbCourse as
        select C.cID, C.cTitle, C.cHours
        from   (trainingCourse C join technologyArea A
               on areaID=aID)
        where  A.aTitle = 'Database';
```

This revision of the standard also allowed for a *natural join* specification in the `from` clause. A natural join is a shorthand specification of an equality join where the attributes having the same name in both tables are joined such that

they are equal and only one copy of the attribute is included in the result. Recall query Q2, which finds those employees who have taken a training course in the database technology area. The query Q2A specified the same query using the `in` set membership comparison operator in the `where` clause. The query can also be specified using natural joined tables in the `from` clause. Query Q2B answers the same query by performing a natural join of the `employee`, `takes` and `dbCourse` tables. The `takes` and `dbCourse` tables are natural joined on the `cID` attribute and the result is natural joined with the `employee` table on the `eID` attribute:

```
Q2B:    select  distinct E.eID, E.eLast, E.eFirst, E.eTitle
        from    (employee E natural join
                (takes T natural join dbCourse D));
```

Obviously, the natural join is based on the attributes having the same name. The `as` construct allows for the renaming of tables and attributes in the `from` clause to facilitate a natural join instead of an explicit join. Query Q1D renames the `technologyArea` table to the `area` table with its `aID` attribute renamed to `areaID` to perform a natural join:

```
Q1D:    create view dbCourse as
        select cID, cTitle, cHours
        from   (trainingCourse natural join
               (technologyArea as
               area(areaID, aTitle, aURL, aLeadID)))
        where  aTitle = 'Database';
```

All of the basic SQL queries with the joined conditions specified in the `where` clause are still valid SQL queries. The joined tables capability provides an alternative specification of the query.

5.5 QUERY OPTIMIZATION

In the procedural relational algebra language, the concept of query optimization was introduced to motivate that alternative query specifications may result in varying performance. Heuristics were discussed to perform selections as early as possible and to reorder joins using the theoretical properties of the relational operators to find an alternative specification of the query that was more efficient. Query optimization is also important in declarative languages, although the coverage of the declarative query languages (DRC, TRC, and SQL) thus far has emphasized the correct specification of a query. After all, the declarative query languages specify what data to retrieve and not how to retrieve the data. However, it is this declarative specification of *what* data to retrieve instead of *how* to retrieve the data that leads to the concept of query optimization for SQL.

There are obviously different ways to specify the characteristics of the data to be retrieved in SQL, and these alternative query specifications may result in query executions with different performance characteristics. A review of the various specifications for the query Q2, which finds the employees who took a training course offered in the database technology area, illustrates some of the differences in the query evaluation.

The original specification of Q2 uses a nested correlated subquery to illustrate the exists conditional and the similarity of SQL to TRC:

```
√Q2:  select  E.eID, E.eLast, E.eFirst, E.eTitle
      from    employee E
      where   exists
              (select *
              from    takes T, dbCourse D
              where   T.eID=E.eID and T.cID=D.cID);
```

Since the inner query is correlated on the employee eID, the nested query must be executed for each employee tuple.

The Q2A specification uses a nested uncorrelated query to illustrate the use of the in set membership conditional:

```
√Q2A: select  E.eID, E.eLast, E.eFirst, E.eTitle
      from    employee E
      where   E.eID in
              (select T.eID
              from    takes T, dbCourse D
              where   T.cID=D.cID);
```

Since the nested query is not correlated, the subquery can be evaluated once, b each employee ID value is checked for membership in the result of the inner quer

The Q2B specification illustrates the natural join option in the from clause:

```
Q2B:  select  distinct E.eID, E.eLast, E.eFirst, E.eTitle
      from    (employee E natural join
              (takes T natural join dbCourse D));
```

which is the variation of another specification of the query Q2C that incorpora the join conditions in the where clause:

```
√Q2C: select  distinct E.eID, E.eLast, E.eFirst, E.eTitle
      from    employee E, takes T, dbCourse D
      where   E.eID = T.eID and T.cID=D.cID;
```


Both of these unnested versions of Q2 require the removal of duplicates using distinct, since an employee can be included in the result for each database training course the employee took.

To summarize, some of the alternatives for specifying query Q2 include

- nested correlated subquery using an existential conditional
- nested uncorrelated subquery using the set membership comparison
- unnested query with joined tables in the from clause requiring distinct to remove duplicates.
- unnested query with join specified in the where clause requiring distinct to remove duplicates

In general terms, an unnested version of a query is usually more efficient than a nested version. Similarly, a nested uncorrelated specification is generally evaluated more efficiently than a correlated one. Also, using joined tables in the from clause may be evaluated more efficiently than a corresponding join specified in the where clause.

Although query optimization is the responsibility of the database system, the performance of the database application is ultimately the responsibility of the database administrator (DBA). If the performance of certain queries is not appropriate, the DBA is responsible for tuning the database, based on guidelines for database administration provided in the manual for the particular database product employed in the application. Obviously, adding indexes to certain attributes or combination of attributes may improve performance. However, revising the specification of a query may also be necessary. Most database products provide the ability to view the plan for evaluating a query, which can be used in conjunction with vendor guidelines to tune the performance of the database application.

5.6 BEYOND THE QUERY LANGUAGE

Although SQL is short for Structured Query Language, the SQL standard has evolved to include a DDL for defining tables and relationships between tables; a DML for inserting, updating, and deleting tuples from tables; and a DCL for database integrity and security.

5.6.1 Data Definition Language

A DDL is a language used for defining the database. The data definition component of SQL was added to the standard in SQL-92. The following defines the technologyArea table in SQL:

```
create table technologyArea
(   aID      char(3)       not null,
    aTitle   varchar(20)   not null,
    aURL     varchar(50)   default
                           'http://www.company.intranet/technology/',
    aLeadID char(9)        not null,
    constraint areaKey
           primary key (aID),
    constraint areaUniqueTitle
           unique (aTitle),
    constraint areaUniqueLead
           unique (aLeadID),
    constraint areaForeignKey
           foreign key (aLeadID) references employee (eID)
           on update cascade;);
```

The name and data type or domain of each attribute of the table is declared along with any constraints for that attribute. The standard supports scalar data types that include fixed- and varying-length character string data types, fixed- and varying-length bit string data types, and various types for numbers, including `smallint`, `integer`, `decimal`, and `float`. Possible attribute constraints include `not null` and `default` value. The `not null` constraint indicates that a `null` value is not permitted for that attribute. The `default` value constraint uses the default value specified for that attribute when an explicit value is not given when creating a new tuple. In the `technologyArea` table, the URL for the company's main technology Web page is given as the default value for the `aURL` attribute.

The definition of table constraints follows the attribute definitions. Possible table constraints include a primary key constraint, uniqueness constraint, and referential integrity. For the `technologyArea` table, the attribute `aID` forms the primary key, the attributes `aTitle` and `aLeadID` must be unique, and the attribute `aLeadID` is a foreign key that references the primary key `eID` of the `employee` table. Each of these table constraints can be optionally named as shown to allow for explicit deletion if desired. The referential integrity constraint allows for the specification of actions for a delete or an update of the referenced attribute. Since there is no action specified for delete, the database system will not allow the removal of an employee whose identification number is the value of the `aLeadID` attribute for some tuple in the `technologyArea` table. The action specified for an update of the `eID` attribute of the `employee` table that appears as a foreign key in the `technologyArea` table is to automatically cascade this update to the foreign key `aLeadID`. Other referential triggered actions include `set default` to automatically set the foreign key value to a default value for the attribute and `set null` to automatically set the foreign key value to a null value. The action specified depends on the semantics of the enterprise and must be wisely chosen by the database designer.

5.6.2 Data Manipulation Language

DML extends the query language with the capability to insert, update, and delete tuples from tables.

The `insert` statement provides the ability to insert data into a table. The following example illustrates an insert using explicit values to add a row to the `employee` table:

```
I1:    insert into employee values
       ('111', 'Last111', 'First111', 'Data Administrator', 75111);
```

In the example `I1`, the columns of `employee` are assumed to be in a left-to-right ordering of the fields of the table. However, the insert statement also allows an optional attribute list to specify a different ordering of attributes or to take advantage of the specification of default values. Recall that the `create table` statement for `technologyArea` defined a default value for the URL: `'http://www.company.intranet/technology/'`. The following `insert` statement adds a new `technologyArea` tuple for the `'Web'` technology area. Since the `aURL` attribute is not included in the list, the default value for the URL is used:

```
I2:    insert  into technologyArea (aID, aTitle, aLeadID)
       values  ('WW', 'Web', '369');
```

A select statement can also be used as the source of an insert into statement. Assuming that the table `managers` has been defined appropriately by a create table statement, then the `managers` table can be populated with the identification number of employees having the title 'Manager':

```
I3:    insert  into managers
       select  eID
       from    employee
       where   eTitle = 'Manager';
```

In this case, `managers` is an intermediate table, which is populated at the time that the insert statement is executed.

The `update` statement provides the ability to change existing data in a table. The following example illustrates an update that gives employees who lead technology areas a 3% raise:

```
U1:    update  employee
       set     eSalary = eSalary * 1.03
       where   E.eID in
               (select A.aLeadID
               from    technologyArea A);
```

The set clause specifies the attribute and its modified value. In the preceding example, the eSalary attribute is increased by 3%. If additional attributes are to be changed, a comma is used to separate the update specification for each attribute. The where clause specifies the rows of the table to be updated. If all rows are to be modified, then the where clause is not specified. In the example, the where clause restricts the update of the salary to only those employees who lead technology areas.

The delete statement provides the ability to delete data from a table. The following example illustrates the deletion of technology areas that do not offer any training courses:

```
D1:   delete from technologyArea
      where       aID not in
                  (select C.areaID
                  from    trainingCourse C);
```

The where clause restricts the delete operation to the rows that satisfy the where condition. If the where clause is not specified, then all rows of the table are removed. In the foregoing example, the where clause specifies that rows in the technologyArea table will be deleted if the aID attribute value does not appear as an areaID value in any trainingCourse row.

5.6.3 Data Control

SQL also includes components for data control: integrity and security. Both integrity and security are important components of a database system. SQL allows for the specification of general integrity constraints that validate the data in the database for consistency. It also supports the granting and revoking of access privileges for data security.

Integrity. The data integrity component of a database system maintains the correctness of the data according to the semantics of the enterprise as specified in the database implementation. Some common integrity constraint specifications are built in to the DDL, such as attribute and table constraints. Most of these constraints are a shorthand for common integrity constraints that can be specified using a integrity constraint specification language.

SQL provides a create assertion statement for the specification of general integrity constraints, which specifies a check condition that must be TRUE for every database instance. In the following example of a general integrity constraint, assume that an employee lead must have taken at least one course in the technology area that they lead:

```
create  assertion areaLeadAtLeastOne
check   (not exists
            (select *
            from    technologyArea A
            where   A.aLeadID not in
                (select T.eID
                from    takes T, trainingCourse C
                where   T.cID = C.cID and
                        C.areaID=A.aID)));
```


The constraint is violated if the check condition evaluates to FALSE. In this example, the condition is violated if there exists a tuple in the technologyArea table where the value of the aLeadID value does not appear in the multiset of employee IDs that have taken a course in that technology area.

Security. The security component of a database system is responsible for enforcing the access rules, based on the discretionary privileges granted on a database object. SQL incorporates table privileges that are granted to or revoked from an authorized database user on a table. The owner of the database table is the authorized database user who created the table. The table owner is automatically granted all table privileges on the created table. The owner of the table may then use the grant and revoke statements to manage the privileges on the database table.

The table privileges include select, insert, update, delete, and reference. An authorized database user must have the select table privilege to access any column of a table. Inserting data into a table requires the insert table privilege, which can be restricted to specify a list of named columns. Similarly, the update privilege allows a user to update data in a table, and this privilege can be restricted to specify the columns of the table that the authorized user is allowed to update. An authorized database user must have the delete table privilege to delete any tuples from a table. The reference table privilege, which may be restricted to columns of a table, allows the column of a table to be referenced in the specification of an integrity constraint.

The grant statement adds the list of privileges on the given tables to the list of users specified. Assume that there is a training department that is responsible for the technical training courses and enrollment for those courses within the company. The training department can grant the select privilege to humanResources on the takes, trainingCourse, and technologyArea tables, so that the humanResources user can determine the skill set of an employee:

```
grant   select
on      takes, trainingCourse, technologyArea
to      humanResources
```

When granting privileges, the user granting the privileges may optionally specify the with grant option, which allows the users to grant these privileges (or a subset thereof) to other authorized database users. Assume that humanResources is the owner of the employee table. The humanResources user must grant the

reference privilege on the eID attribute of the employee table to the trainingCoordinator so that the employee identification number entered in the takes table can be verified for referential integrity. The with grant option allows the trainingCoordinator to grant this reference privilege to another authorized database user.

```
grant    reference(eID)
on       employee
to       trainingCoordinator
with grant option
```

The revoke statement removes the list of privileges on the given tables from the list of users specified. In the following revoke statement, the training department removes the ability for humanResources to access the takes table:

```
revoke  select
on      takes
from    humanResources
```

Views also provide a mechanism that works in conjunction with privileges for data security. The insert, update, and reference table privileges can be restricted to specify a list of named columns. However, the select table privilege gives the user access to any column of a table. To restrict a user's access to only some of the columns of a table, a view can be defined on that table to project the desired attributes. A grant statement of the select table privilege on the view allows select access to only those columns provided in the view.

5.7 SUMMARY

SQL uses a syntactic variation of TRC to provide a practical and declarative language to query a relational database. This section provides a brief syntax summary of the SQL constructs described in this chapter, including queries, views, data definition, data manipulation (insert, update, and delete), integrity and security. The conventions for the syntax summary use square brackets [. . .] to denote optional items, | to indicate a choice of one of the items specified, and identifiers in SMALL CAPS to denote items that can be replaced by names or further expanded.

The select statement forms the basis of an SQL query, selecting a list of attributes from a list of tables that optionally satisfy a where condition. The optional keyword distinct returns a set of tuples instead of the default multiset of tuples as the result of the query. The optional group by clause allows for the specification of grouping attributes for the application of aggregate functions (e.g., min, max, avg, sum, count) specified in the select clause. An optional having clause allows for the specification of a condition on the result of the grouping. The result of a

query can be optionally ordered using the order by clause, which specifies the columns to be ordered and the sort order (either ascending or descending):

```
select      [distinct] ATTRIBUTE-LIST
from        TABLE-LIST
[where      WHERE-CONDITION]
[group by   GROUPING-ATTRIBUTES
[having     HAVING-CONDITION]]
[order by   COLUMN-NAME [ asc | desc ], ... ]
```

A view or virtual table provides the inherent capability for abstraction, associating the name of the view with a defining select statement, which is executed each time the view name is referenced:

```
create view VIEW-NAME as
SELECT-STATEMENT
```

SQL also includes a DDL that provides the capability to define tables. Each attribute or column is defined to be of a specified type and can have associated attribute constraints, such as not null or default value. The list of table constraints include primary key, uniqueness, and referential integrity (foreign key) constraints:

```
create table TABLE-NAME
(
COLUMN-NAME     COLUMN-TYPE     [ATTRIBUTE-CONSTRAINT],
                ...
[ TABLE-CONSTRAINT-LIST ]
)
```

SQL also provides statements for data manipulation, including insert, update, and delete:

The insert into statement provides a mechanism to insert tuples into the named table according to the specified ATTRIBUTE-LIST using the specified SOURCE:

```
insert into TABLE-NAME [ (ATTRIBUTE-LIST) ]
SOURCE
```

The SOURCE of the insert into statement is either a list of explicit values, as in

```
insert into TABLE-NAME [ (ATTRIBUTE-LIST) ]
values ( EXPLICIT-VALUES )
```

or the specification of a select statement, as in

```
insert into TABLE-NAME [ (ATTRIBUTE-LIST) ]
SELECT-STATEMENT
```

In either case, if a column of TABLE-NAME is not included in the ATTRIBUTE-LIST, then a default value is assigned for that column. Note that if an omitted column does not have a default value specified in the DDL for TABLE-NAME, then the insert results in an error. If ATTRIBUTE-LIST is not specified, then a left-to-right ordering of the attributes within that table is assumed.

The `update` statement modifies the columns given in the `set` clause to the values determined by the expressions specified. The optional `where` clause can restrict the rows of the table to be updated:

```
update  TABLE-NAME
set     COLUMN-NAME = VALUE-EXPRESSION, ...
[where  UPDATE-CONDITION]
```

The `delete` statement removes tuples from the table specified. The optional `where` clause restricts the delete operation to the rows that satisfy the DELETE-CONDITION. If the `where` clause is not specified, then all rows of the table are deleted:

```
delete  from TABLE-NAME
[where  DELETE-CONDITION]
```

Integrity constraints validate the data in the database. SQL provides for the specification of general integrity constraints with the `create assertion` statement which checks that the ASSERTION-CONDITION specified is TRUE on every database instance:

```
create  assertion ASSERTION-NAME
check   (ASSERTION-CONDITION)
```

The security component of SQL incorporates table privileges that are granted to or revoked from an authorized database user on a table. The table privileges include `select`, `insert`, `update`, `delete`, and `reference`. The `grant` statement adds the list of privileges on the given tables to the list of users specified. When granting privileges, the user granting the privileges may optionally specify the `with grant option`, which allows the users in the USER-LIST to grant these privileges (or a subset thereof) to other authorized database users:

```
grant   PRIVILEGE-LIST
on      TABLE-COLUMN-LIST
to      USER-LIST
[with grant option]
```


The `revoke` statement removes the list of privileges on the given tables from the list of users specified:

```
revoke  PRIVILEGE-LIST
on      TABLE-COLUMN-LIST
from    USER-LIST
```

EXERCISES

✓ Use the educational tool to *check* the answers to these queries.

✓**5.1** Answer the following queries in SQL over the EMPLOYEE TRAINING schema:

```
employee(eID, eLast, eFirst, eTitle, eSalary)
technologyArea(aID, aTitle, aURL, aLeadID)
trainingCourse(cID, cTitle, cHours, areaID)
takes(eID, cID, tYear, tMonth, tDay)
```

(a) Which employees took the 'Enterprise JavaBeans' training course?

```
(eID, eLast, eFirst, eTitle)
```

(b) Which coaches have not taken a training course in the 'Java' technology area?

```
(eID, eLast, eFirst)
```

(c) Which employees took *more than one* course in the 'Database' technology area?

```
(eID, eLast, eFirst, eTitle)
```

(d) Who is the lowest-paid manager?

```
(eID, eLast, eFirst, eTitle, eSalary)
```

(e) What are the titles of employees who earn more than a 'Coach'?

```
(eTitle)
```

(f) Which leads have taken *all* of the training courses in the technology area that they lead?

```
(eID, eLast, eFirst, eTitle)
```