Step	By	Action
1	A	GRANT p TO B, E WITH GRANT OPTION
2	B	GRANT p TO C WITH GRANT OPTION
3	C	GRANT p TO D WITH GRANT OPTION
4	E	GRANT p TO C
5	E	GRANT p TO D WITH GRANT OPTION
6	A	REVOKE GRANT OPTION FOR p FROM B CASCADE

Figure 10.7: Sequence of actions for Exercise 10.1.3

! **Exercise 10.1.4:** Show the final grant diagram after the following steps, assuming A is the owner of the relation to which privilege p refers.

Step	By	Action
1	A	GRANT p TO B WITH GRANT OPTION
2	B	GRANT p TO B WITH GRANT OPTION
3	A	REVOKE p FROM B CASCADE

10.2 Recursion in SQL

The SQL-99 standard includes provision for recursive definitions of queries. Although this feature is not part of the "core" SQL-99 standard that every DBMS is expected to implement, at least one major system — IBM's DB2 — does implement the SQL-99 proposal, which we describe in this section.

10.2.1 Defining Recursive Relations in SQL

The WITH statement in SQL allows us to define temporary relations, recursive or not. To define a recursive relation, the relation can be used within the WITH statement itself. A simple form of the WITH statement is:

WITH R AS <definition of R> <query involving R>

That is, one defines a temporary relation named R, and then uses R in some query. The temporary relation is not available outside the query that is part of the WITH statement.

More generally, one can define several relations after the WITH, separating their definitions by commas. Any of these definitions may be recursive. Several defined relations may be mutually recursive; that is, each may be defined in terms of some of the other relations, optionally including itself. However, any relation that is involved in a recursion must be preceded by the keyword RECURSIVE. Thus, a more general form of WITH statement is shown in Fig. 10.8.

Example 10.8: Many examples of the use of recursion can be found in a study of paths in a graph. Figure 10.9 shows a graph representing some flights of two

```
WITH
   [RECURSIVE] R1 AS <definition of R1>,
   [RECURSIVE] R2 AS <definition of R2>,
     ...
   [RECURSIVE] Rn AS <definition of Rn>
<query involving R1, R2, ... , Rn >
```

Figure 10.8: Form of a `WITH` statement defining several temporary relations

hypothetical airlines — *Untried Airlines* (UA), and *Arcane Airlines* (AA) — among the cities San Francisco, Denver, Dallas, Chicago, and New York. The data of the graph can be represented by a relation

```
Flights(airline, frm, to, departs, arrives)
```

and the particular tuples in this table are shown in Fig. 10.9.

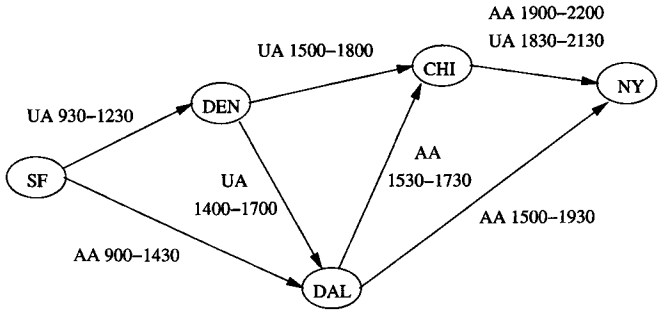

Figure 10.9: A map of some airline flights

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Figure 10.10: Tuples in the relation `Flights`

The simplest recursive question we can ask is "For what pairs of cities (x, y) is it possible to get from city x to city y by taking one or more flights?" Before

writing this query in recursive SQL, it is useful to express the recursion in the Datalog notation of Section 5.3. Since many concepts involving recursion are easier to express in Datalog than in SQL, you may wish to review the terminology of that section before proceeding. The following two Datalog rules describe a relation `Reaches(x,y)` that contains exactly these pairs of cities.

1. `Reaches(x,y)` ← `Flights(a,x,y,d,r)`
2. `Reaches(x,y)` ← `Reaches(x,z) AND Reaches(z,y)`

The first rule says that `Reaches` contains those pairs of cities for which there is a direct flight from the first to the second; the airline a, departure time d, and arrival time r are arbitrary in this rule. The second rule says that if you can reach from city x to city z and you can reach from z to city y, then you can reach from x to y.

Evaluating a recursive relation requires that we apply the Datalog rules repeatedly, starting by assuming there are no tuples in `Reaches`. We begin by using Rule (1) to get the following pairs in `Reaches`: (`SF`, `DEN`), (`SF`, `DAL`), (`DEN`, `CHI`), (`DEN`, `DAL`), (`DAL`, `CHI`), (`DAL`, `NY`), and (`CHI`, `NY`). These are the seven pairs represented by arcs in Fig. 10.9.

In the next round, we apply the recursive Rule (2) to put together pairs of arcs such that the head of one is the tail of the next. That gives us the additional pairs (`SF`, `CHI`), (`DEN`, `NY`), and (`SF`, `NY`). The third round combines all one- and two-arc pairs together to form paths of length up to four arcs. In this particular diagram, we get no new pairs. The relation `Reaches` thus consists of the ten pairs (x, y) such that y is reachable from x in the diagram of Fig. 10.9. Because of the way we drew the diagram, these pairs happen to be exactly those (x, y) such that y is to the right of x in Fig 10.9.

From the two Datalog rules for `Reaches` in Example 10.8, we can develop a SQL query that produces the relation `Reaches`. This SQL query places the Datalog rules for `Reaches` in a `WITH` statement, and follows it by a query. In our example, the desired result was the entire `Reaches` relation, but we could also ask some query about `Reaches`, for instance the set of cities reachable from Denver.

```
1)  WITH RECURSIVE Reaches(frm, to) AS
2)          (SELECT frm, to FROM Flights)
3)      UNION
4)          (SELECT R1.frm, R2.to
5)           FROM Reaches R1, Reaches R2
6)           WHERE R1.to = R2.frm)
7)  SELECT * FROM Reaches;
```

Figure 10.11: Recursive SQL query for pairs of reachable cities

Figure 10.11 shows how to express `Reaches` as a SQL query. Line (1) introduces the definition of `Reaches`, while the actual definition of this relation is in

Mutual Recursion

There is a graph-theoretic way to check whether two relations or predicates are mutually recursive. Construct a *dependency graph* whose nodes correspond to the relations (or predicates if we are using Datalog rules). Draw an arc from relation A to relation B if the definition of B depends directly on the definition of A. That is, if Datalog is being used, then A appears in the body of a rule with B at the head. In SQL, A would appear in a `FROM` clause, somewhere in the definition of B, possibly in a subquery. If there is a cycle involving nodes R and S, then R and S are *mutually recursive*. The most common case will be a loop from R to R, indicating that R depends recursively upon itself.

lines (2) through (6).

That definition is a union of two queries, corresponding to the two Datalog rules by which `Reaches` was defined. Line (2) is the first term of the union and corresponds to the first, or basis rule. It says that for every tuple in the `Flights` relation, the second and third components (the `frm` and `to` components) are a tuple in `Reaches`.

Lines (4) through (6) correspond to Rule (2), the recursive rule, in the definition of `Reaches`. The two `Reaches` subgoals in Rule (2) are represented in the `FROM` clause by two aliases `R1` and `R2` for `Reaches`. The first component of `R1` corresponds to x in Rule (2), and the second component of `R2` corresponds to y. Variable z is represented by both the second component of `R1` and the first component of `R2`; note that these components are equated in line (6).

Finally, line (7) describes the relation produced by the entire query. It is a copy of the `Reaches` relation. As an alternative, we could replace line (7) by a more complex query. For instance,

```
7)  SELECT to FROM Reaches WHERE frm = 'DEN';
```

would produce all those cities reachable from Denver. □

10.2.2 Problematic Expressions in Recursive SQL

The SQL standard for recursion does not allow an arbitrary collection of mutually recursive relations to be written in a `WITH` clause. There is a small matter that the standard requires only that *linear* recursion be supported. A linear recursion, in Datalog terms, is one in which no rule has more than one subgoal that is mutually recursive with the head. Notice that Rule (2) in Example 10.8 has two subgoals with predicate `Reaches` that are mutually recursive with the head (a predicate is always mutually recursive with itself; see the box on Mutual

Recursion). Thus, technically, a DBMS might refuse to execute Fig. 10.11 and yet conform to the standard.[1]

But there is a more important restriction on SQL recursions, one that, if violated leads to recursions that cannot be executed by the query processor in any meaningful way. To be a legal SQL recursion, the definition of a recursive relation R may involve only the use of a mutually recursive relation S (including R itself) if that use is "monotone" in S. A use of S is *monotone* if adding an arbitrary tuple to S might add one or more tuples to R, or it might leave R unchanged, but it can never cause any tuple to be deleted from R. The following example suggests what can happen if the monotonicity requirement is not respected.

Example 10.9: Suppose relation R is a unary (one-attribute) relation, and its only tuple is (0). R is used as an EDB relation in the following Datalog rules:

1. `P(x) ← R(x) AND NOT Q(x)`
2. `Q(x) ← R(x) AND NOT P(x)`

Informally, the two rules tell us that an element x in R is either in P or in Q but not both. Notice that P and Q are mutually recursive.

If we start out, assuming that both P and Q are empty, and apply the rules once, we find that $P = \{(0)\}$ and $Q = \{(0)\}$; that is, (0) is in both IDB relations. On the next round, we apply the rules to the new values for P and Q again, and we find that now both are empty. This cycle repeats as long as we like, but we never converge to a solution.

In fact, there are two "solutions" to the Datalog rules:

a) $P = \{(0)\}$ $Q = \emptyset$
b) $P = \emptyset$ $Q = \{(0)\}$

However, there is no reason to assume one over the other, and the simple iteration we suggested as a way to compute recursive relations never converges to either. Thus, we cannot answer a simple question such as "Is $P(0)$ true?"

The problem is not restricted to Datalog. The two Datalog rules of this example can be expressed in recursive SQL. Figure 10.12 shows one way of doing so. This SQL does not adhere to the standard, and no DBMS should execute it. □

The problem in Example 10.9 is that the definitions of P and Q in Fig. 10.12 are not monotone. Look at the definition of P in lines (2) through (5) for instance. P depends on Q, with which it is mutually recursive, but adding a tuple to Q can delete a tuple from P. Notice that if $R = \{(0)\}$ and Q is empty, then $P = \{(0)\}$. But if we add (0) to Q, then we delete (0) from P. Thus, the definition of P is not monotone in Q, and the SQL code of Fig. 10.12 does not meet the standard.

[1] Note, however, that we can replace either one of the uses of **Reaches** in line (5) of Fig. 10.11 by **Flights**, and thus make the recursion linear. Nonlinear recursions can frequently — although not always — be made linear in this fashion.

```
 1)  WITH
 2)      RECURSIVE P(x) AS
 3)              (SELECT * FROM R)
 4)          EXCEPT
 5)              (SELECT * FROM Q),

 6)      RECURSIVE Q(x) AS
 7)              (SELECT * FROM R)
 8)          EXCEPT
 9)              (SELECT * FROM P)

10)  SELECT * FROM P;
```

Figure 10.12: Query with nonmonotonic behavior, illegal in SQL

Example 10.10: Aggregation can also lead to nonmonotonicity. Suppose we have unary (one-attribute) relations P and Q defined by the following two conditions:

1. P is the union of Q and an EDB relation R.
2. Q has one tuple that is the sum of the members of P.

We can express these conditions by a `WITH` statement, although this statement violates the monotonicity requirement of SQL. The query shown in Fig. 10.13 asks for the value of P.

```
 1)  WITH
 2)      RECURSIVE P(x) AS
 3)              (SELECT * FROM R)
 4)          UNION
 5)              (SELECT * FROM Q),

 6)      RECURSIVE Q(x) AS
 7)          SELECT SUM(x) FROM P

 8)  SELECT * FROM P;
```

Figure 10.13: Nonmonotone query involving aggregation, illegal in SQL

Suppose that R consists of the tuples (12) and (34), and initially P and Q are both empty. Figure 10.14 summarizes the values computed in the first six rounds. Note that both relations are computed, in one round, from the values of the relations at the previous round. Thus, P is computed in the first round

Round	P	Q
1)	{(12), (34)}	{NULL}
2)	{(12), (34), NULL}	{(46)}
3)	{(12), (34), (46)}	{(46)}
4)	{(12), (34), (46)}	{(92)}
5)	{(12), (34), (92)}	{(92)}
6)	{(12), (34), (92)}	{(138)}

Figure 10.14: Iterative calculation for a nonmonotone aggregation

to be the same as R, and Q is {NULL}, since the old, empty value of P is used in line (7).

At the second round, the union of lines (3) through (5) is the set

$$R \cup \{\texttt{NULL}\} = \{(12),\ (34),\ \texttt{NULL}\}$$

so that set becomes the new value of P. The old value of P was {(12), (34)}, so on the second round $Q = \{(46)\}$. That is, 46 is the sum of 12 and 34.

At the third round, we get $P = \{(12), (34), (46)\}$ at lines (2) through (5). Using the old value of P, {(12), (34), NULL}, Q is defined by lines (6) and (7) to be {(46)} again. Remember that NULL is ignored in a sum.

At the fourth round, P has the same value, {(12), (34), (46)}, but Q gets the value {(92)}, since 12+34+46=92. Notice that Q has lost the tuple (46), although it gained the tuple (92). That is, adding the tuple (46) to P has caused a tuple (by coincidence the same tuple) to be deleted from Q. That behavior is the nonmonotonicity that SQL prohibits in recursive definitions, confirming that the query of Fig. 10.13 is illegal. In general, at the $2i$th round, P will consist of the tuples (12), (34), and $(46i - 46)$, while Q consists only of the tuple $(46i)$. □

10.2.3 Exercises for Section 10.2

Exercise 10.2.1: The relation

```
Flights(airline, frm, to, departs, arrives)
```

from Example 10.8 has arrival- and departure-time information that we did not consider. Suppose we are interested not only in whether it is possible to reach one city from another, but whether the journey has reasonable connections. That is, when using more than one flight, each flight must arrive at least an hour before the next flight departs. You may assume that no journey takes place over more than one day, so it is not necessary to worry about arrival close to midnight followed by a departure early in the morning.

a) Write this recursion in Datalog.

b) Write the recursion in SQL.

! **Exercise 10.2.2:** In Example 10.8 we used `frm` as an attribute name. Why did we not use the more obvious name `from`?

Exercise 10.2.3: Suppose we have a relation

```
SequelOf(movie, sequel)
```

that gives the immediate sequels of a movie, of which there can be more than one. We want to define a recursive relation `FollowOn` whose pairs (x, y) are movies such that y was either a sequel of x, a sequel of a sequel, or so on.

a) Write the definition of `FollowOn` as recursive Datalog rules.

b) Write the definition of `FollowOn` as a SQL recursion.

c) Write a recursive SQL query that returns the set of pairs (x, y) such that movie y is a follow-on to movie x, but is not a sequel of x.

d) Write a recursive SQL query that returns the set of pairs (x, y) meaning that y is a follow-on of x, but is neither a sequel nor a sequel of a sequel.

! e) Write a recursive SQL query that returns the set of movies x that have at least two follow-ons. Note that both could be sequels, rather than one being a sequel and the other a sequel of a sequel.

! f) Write a recursive SQL query that returns the set of pairs (x, y) such that movie y is a follow-on of x but y has at most one follow-on.

Exercise 10.2.4: Suppose we have a relation

```
Rel(class, rclass, mult)
```

that describes how one ODL class is related to other classes. Specifically, this relation has tuple (c, d, m) if there is a relation from class c to class d. This relation is multivalued if $m =$ `'multi'` and it is single-valued if $m =$ `'single'`. It is possible to view `Rel` as defining a graph whose nodes are classes and in which there is an arc from c to d labeled m if and only if (c, d, m) is a tuple of `Rel`. Write a recursive SQL query that produces the set of pairs (c, d) such that:

a) There is a path from class c to class d in the graph described above.

b) There is a path from c to d along which every arc is labeled `single`.

! c) There is a path from c to d along which at least one arc is labeled `multi`.

d) There is a path from c to d but no path along which all arcs are labeled `single`.