Note 3.3 SQL AND MULTISET RELATIONAL ALGEBRA - PART 3

Unlike the SQL set and aggregation operations that we studied earlier in this chapter, SQL subqueries do not have directly equivalent operations in the relational algebra. Most SQL queries involving subqueries can be rewritten in a way that does not require the use of subqueries, and thus they have equivalent relational algebra expressions.

Rewriting to relational algebra can benefit from two extended relational algebra operations called *semijoin*, denoted $\ltimes$, and *antijoin*, denoted $\overline{\ltimes}$, which are supported internally by many database implementations (the symbol $\triangleright$ is sometimes used in place of $\overline{\ltimes}$ to denote antijoin). For example, given relations r and s, $r \ltimes_{r.A=s.B} s$ outputs all tuples in r that have at least one tuple in s whose $s.B$ attribute value matches that tuples $r.A$ attribute value. Conversely, $r\ \overline{\ltimes}_{r.A=s.B}\ s$ outputs all tuples in r that have do not have any such matching tuple in s. These operators can be used to rewrite many subqueries that use the **exists** and **not exists** connectives.

Semijoin and antijoin can be expressed using other relational algebra operations, so they do not add any expressive power, but they are nevertheless quite useful in practice since they can be implemented very efficiently.

However, the process of rewriting SQL queries that contain subqueries is in general not straightforward. Database system implementations therefore extend the relational algebra by allowing σ and Π operators to invoke subqueries in their predicates and projection lists.

3.9 Modification of the Database

We have restricted our attention until now to the extraction of information from the database. Now, we show how to add, remove, or change information with SQL.

3.9.1 Deletion

A **delete** request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by:

```
delete from r
where P;
```

where P represents a predicate and r represents a relation. The **delete** statement first finds all tuples t in r for which $P(t)$ is true, and then deletes them from r. The **where** clause can be omitted, in which case all tuples in r are deleted.

Note that a **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation. The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the **where** clause may be empty. The request:

```
delete from instructor;
```

deletes all tuples from the *instructor* relation. The *instructor* relation itself still exists, but it is empty.

Here are examples of SQL delete requests:

- Delete all tuples in the *instructor* relation pertaining to instructors in the Finance department.

```
delete from instructor
where dept_name = 'Finance';
```

- Delete all instructors with a salary between $13,000 and $15,000.

```
delete from instructor
where salary between 13000 and 15000;
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

This **delete** request first finds all departments located in Watson and then deletes all *instructor* tuples pertaining to those departments.

Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** nested in the **where** clause of a **delete**. The **delete** request can contain a nested **select** that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all instructors with salary below the average at the university. We could write:

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

The **delete** statement first tests each tuple in the relation *instructor* to check whether the salary is less than the average salary of instructors in the university. Then, all tuples that pass the test—that is, represent an instructor with a lower-than-average salary—are deleted. Performing all the tests before performing any deletion is important—if some tuples are deleted before other tuples have been tested, the average salary may change, and the final result of the **delete** would depend on the order in which the tuples were processed!

3.9.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes.

The simplest **insert** statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title "Database Systems" and four credit hours. We write:

```
insert into course
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

In this example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. For the benefit of users who may not remember the order of the attributes, SQL allows the attributes to be specified as part of the **insert** statement. For example, the following SQL **insert** statements are identical in function to the preceding one:

```
insert into course (course_id, title, dept_name, credits)
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

insert into course (title, course_id, credits, dept_name)
        values ('Database Systems', 'CS-437', 4, 'Comp. Sci.');
```

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000. We write:

```
insert into instructor
        select ID, name, dept_name, 18000
        from student
        where dept_name = 'Music' and tot_cred > 144;
```

Instead of specifying a tuple as we did earlier in this section, we use a **select** to specify a set of tuples. SQL evaluates the **select** statement first, giving a set of tuples that is then inserted into the *instructor* relation. Each tuple has an *ID*, a *name*, a *dept_name* (Music), and a salary of $18,000.

It is important that the system evaluate the **select** statement fully before it performs any insertions. If it were to carry out some insertions while the **select** statement was being evaluated, a request such as:

```
insert into student
    select *
    from student;
```

might insert an infinite number of tuples, if the primary key constraint on *student* were absent. Without the primary key constraint, the request would insert the first tuple in *student* again, creating a second copy of the tuple. Since this second copy is part of *student* now, the **select** statement may find it, and a third copy would be inserted into *student*. The **select** statement may then find this third copy and insert a fourth copy, and so on, forever. Evaluating the **select** statement completely before performing insertions avoids such problems. Thus, the above **insert** statement would simply duplicate every tuple in the *student* relation if the relation did not have a primary key constraint.

Our discussion of the **insert** statement considered only examples in which a value is given for every attribute in inserted tuples. It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by *null*. Consider the request:

```
insert into student
    values ('3003', 'Green', 'Finance', null);
```

The tuple inserted by this request specified that a student with *ID* “3003” is in the Finance department, but the *tot_cred* value for this student is not known.

Most relational database products have special “bulk loader” utilities to insert a large set of tuples into a relation. These utilities allow data to be read from formatted text files, and they can execute much faster than an equivalent sequence of insert statements.

3.9.3 Updates

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used. As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query.

Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent. We write:

```
update instructor
set salary= salary * 1.05;
```

The preceding update statement is applied once to each of the tuples in the *instructor* relation.

If a salary increase is to be paid only to instructors with a salary of less than $70,000, we can write:

```
update instructor
set salary = salary * 1.05
where salary < 70000;
```

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **select**s). As with **insert** and **delete**, a nested **select** within an **update** statement may reference the relation that is being updated. As before, SQL first tests all tuples in the relation to see whether they should be updated, and it carries out the updates afterward. For example, we can write the request "Give a 5 percent salary raise to instructors whose salary is less than average" as follows:

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

Let us now suppose that all instructors with salary over $100,000 receive a 3 percent raise, whereas all others receive a 5 percent raise. We could write two **update** statements:

```
update instructor
set salary = salary * 1.03
where salary > 100000;

update instructor
set salary = salary * 1.05
where salary <= 100000;
```

Note that the order of the two **update** statements is important. If we changed the order of the two statements, an instructor with a salary just under $100,000 would receive a raise of over 8 percent.

SQL provides a **case** construct that we can use to perform both updates with a single **update** statement, avoiding the problem with the order of updates.

```
update instructor
set salary = case
                 when salary <= 100000 then salary * 1.05
                 else salary * 1.03
             end
```

The general form of the case statement is as follows:

```
case
    when pred_1 then result_1
    when pred_2 then result_2
    ...
    when pred_n then result_n
    else result_0
end
```

The operation returns $result_i$, where i is the first of $pred_1, pred_2, \ldots, pred_n$ that is satisfied; if none of the predicates is satisfied, the operation returns $result_0$. Case statements can be used in any place where a value is expected.

Scalar subqueries are useful in SQL update statements, where they can be used in the **set** clause. We illustrate this using the *student* and *takes* relations that we introduced in Chapter 2. Consider an update where we set the *tot_cred* attribute of each *student* tuple to the sum of the credits of courses successfully completed by the student. We assume that a course is successfully completed if the student has a grade that is neither 'F' nor null. To specify this update, we need to use a subquery in the **set** clause, as shown below:

```
update student
set tot_cred = (
    select sum(credits)
    from takes, course
    where student.ID= takes.ID and
              takes.course_id = course.course_id and
              takes.grade <> 'F' and
              takes.grade is not null);
```

In case a student has not successfully completed any course, the preceding statement would set the *tot_cred* attribute value to null. To set the value to 0 instead, we could use another **update** statement to replace null values with 0; a better alternative is to replace the clause "**select sum**(*credits*)" in the preceding subquery with the following **select** clause using a **case** expression:

```
select case
    when sum(credits) is not null then sum(credits)
    else 0
    end
```