C H A P T E R 6

Integrity and Security

Exercises

6.1 Answer:

create table *loan*
(*loan-number* char(10),
branch-name char(15),
amount integer,
primary key (*loan-number*),
foreign key (*branch-name*) **references** *branch*)

create table *borrower*
(*customer-name* char(20),
loan-number char(10),
primary key (*customer-name, loan-number*),
foreign key (*customer-name*) **references** *customer*,
foreign key (*loan-number*) **references** *loan*)

Declaring the pair *customer-name, loan-number* of relation *borrower* as primary key ensures that the relation does not contain duplicates.

6.2 Answer:

```
create table employee
    (person-name   char(20),
     street        char(30),
     city          char(30),
     primary key (person-name) )
```

```
create table works
    (person-name   char(20),
     company-name  char(15),
     salary        integer,
     primary key (person-name),
     foreign key (person-name) references employee,
     foreign key (company-name) references company)
```

```
create table company
    (company-name char(15),
     city         char(30),
     primary key (company-name))
```

```
create table manages
    (person-name   char(20),
     manager-name  char(20),
     primary key (person-name),
     foreign key (person-name) references employee,
     foreign key (manager-name) references employee)
```

Note that alternative datatypes are possible.

6.4 **Answer:** The tuples of all employees of the manager, at all levels, get deleted as well! This happens in a series of steps. The initial deletion will trigger deletion of all the tuples corresponding to direct employees of the manager. These deletions will in turn cause deletions of second level employee tuples, and so on, till all direct and indirect employee tuples are deleted.

6.6 **Answer:** The assertion-name is arbitrary. We have chosen the name *perry*. Note that since the assertion applies only to the Perryridge branch we must restrict attention to only the Perryridge tuple of the *branch* relation rather than writing a constraint on the entire relation.

```
create assertion perry check
    (not exists (select *
                 from branch
                 where branch-name = 'Perryridge' and
                       assets ≠ (select sum (amount)
                                 from loan
                                 where branch-name = 'Perryridge')))
```

6.7 Answer:

create trigger *check-delete-trigger* **after delete on** *account*
referencing old row as *orow*
for each row
delete from *depositor*
where *depositor.customer-name* **not in**
(**select** *customer-name* **from** *depositor*
where *account-number* <> *orow.account-number*)
end

6.8 Answer: For inserting into the materialized view *branch-cust* we must set a database trigger on an insert into *depositor* and *account*. We assume that the database system uses *immediate* binding for rule execution. Further, assume that the current version of a relation is denoted by the relation name itself, while the set of newly inserted tuples is denoted by qualifying the relation name with the prefix – **inserted**.

The active rules for this insertion are given below –

define trigger *insert_into_branch-cust_via_depositor*
after insert on *depositor*
referencing new table as *inserted* **for each statement**
insert into *branch-cust*
select *branch-name, customer-name*
from *inserted, account*
where *inserted.account-number = account.account-number*

define trigger *insert_into_branch-cust_via_account*
after insert on *account*
referencing new table as *inserted* **for each statement**
insert into *branch-cust*
select *branch-name, customer-name*
from *depositor, inserted*
where *depositor.account-number = inserted.account-number*

Note that if the execution binding was *deferred* (instead of immediate), then the result of the join of the set of new tuples of *account* with the set of new tuples of *depositor* would have been inserted by *both* active rules, leading to duplication of the corresponding tuples in *branch-cust*.

The deletion of a tuple from *branch-cust* is similar to insertion, except that a deletion from either *depositor* or *account* will cause the natural join of these relations to have a lesser number of tuples. We denote the newly deleted set of tuples by qualifying the relation name with the keyword **deleted**.

```
define trigger delete_from_branch-cust_via_depositor
after delete on depositor
referencing old table as deleted for each statement
delete from branch-cust
    select branch-name, customer-name
    from deleted, account
    where deleted.account-number = account.account-number

define trigger delete_from_branch-cust_via_account
after delete on account
referencing old table as deleted for each statement
delete from branch-cust
    select branch-name, customer-name
    from depositor, deleted
    where depositor.account-number = deleted.account-number
```

6.12 Answer: Usually, a well-designed view and security mechanism can avoid conflicts between ease of access and security. However, as the following example shows, the two purposes do conflict in case the mechanisms are not designed carefully.

Suppose we have a database of employee data and a user whose view involves employee data for employees earning less than $10,000. If this user inserts employee Jones, whose salary is $9,000, but accidentally enters $90,000, several existing database systems will accept this update as a valid update through a view. However, the user will be denied access to delete this erroneous tuple by the security mechanism.

6.16 Answer: A scheme for storing passwords would be to encrypt each password. The user-id can be used to access the encrypted password. An index can be used if the number of users is very large. The password being used in a login attempt is then encrypted and compared with the stored encryption of the correct password. An advantage of this scheme is that passwords are not stored in clear text and the code for decryption need not even exist.