# Chapter 7

# Constraints and Triggers

In this chapter we shall cover those aspects of SQL that let us create "active" elements. An *active* element is an expression or statement that we write once and store in the database, expecting the element to execute at appropriate times. The time of action might be when a certain event occurs, such as an insertion into a particular relation, or it might be whenever the database changes so that a certain boolean-valued condition becomes true.

One of the serious problems faced by writers of applications that update the database is that the new information could be wrong in a variety of ways. For example, there are often typographical or transcription errors in manually entered data. We could write application programs in such a way that every insertion, deletion, and update command has associated with it the checks necessary to assure correctness. However, it is better to store these checks in the database, and have the DBMS administer the checks. In this way, we can be sure a check will not be forgotten, and we can avoid duplication of work.

SQL provides a variety of techniques for expressing *integrity constraints* as part of the database schema. In this chapter we shall study the principal methods. We have already seen key constraints, where an attribute or set of attributes is declared to be a key for a relation. SQL supports a form of referential integrity, called a "foreign-key constraint," the requirement that a value in an attribute or attributes of one relation must also appear as a value in an attribute or attributes of another relation. SQL also allows constraints on attributes, constraints on tuples, and interrelation constraints called "assertions." Finally, we discuss "triggers," which are a form of active element that is called into play on certain specified events, such as insertion into a specific relation.

## 7.1 Keys and Foreign Keys

Recall from Section 2.3.6 that SQL allows us to define an attribute or attributes to be a key for a relation with the keywords `PRIMARY KEY` or `UNIQUE`. SQL also uses the term "key" in connection with certain referential-integrity constraints.

These constraints, called "foreign-key constraints," assert that a value appearing in one relation must also appear in the primary-key component(s) of another relation.

### 7.1.1 Declaring Foreign-Key Constraints

A foreign key constraint is an assertion that values for certain attributes must make sense. Recall, for instance, that in Example 2.21 we considered how to express in relational algebra the constraint that the producer "certificate number" for each movie was also the certificate number of some executive in the `MovieExec` relation.

In SQL we may declare an attribute or attributes of one relation to be a *foreign key*, referencing some attribute(s) of a second relation (possibly the same relation). The implication of this declaration is twofold:

1. The referenced attribute(s) of the second relation must be declared `UNIQUE` or the `PRIMARY KEY` for their relation. Otherwise, we cannot make the foreign-key declaration.

2. Values of the foreign key appearing in the first relation must also appear in the referenced attributes of some tuple. More precisely, let there be a foreign-key $F$ that references set of attributes $G$ of some relation. Suppose a tuple $t$ of the first relation has non-`NULL` values in all the attributes of $F$; call the list of $t$'s values in these attributes $t[F]$. Then in the referenced relation there must be some tuple $s$ that agrees with $t[F]$ on the attributes $G$. That is, $s[G] = t[F]$.

As for primary keys, we have two ways to declare a foreign key.

a) If the foreign key is a single attribute we may follow its name and type by a declaration that it "references" some attribute (which must be a key — primary or unique) of some table. The form of the declaration is

`REFERENCES` <table>(<attribute>)

b) Alternatively, we may append to the list of attributes in a `CREATE TABLE` statement one or more declarations stating that a set of attributes is a foreign key. We then give the table and its attributes (which must be a key) to which the foreign key refers. The form of this declaration is:

`FOREIGN KEY` (<attributes>) `REFERENCES` <table>(<attributes>)

**Example 7.1:** Suppose we wish to declare the relation

```
Studio(name, address, presC#)
```

whose primary key is name and which has a foreign key presC# that references cert# of relation

```
MovieExec(name, address, cert#, netWorth)
```

We may declare presC# directly to reference cert# as follows:

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
);
```

An alternative form is to add the foreign key declaration separately, as

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

Notice that the referenced attribute, cert# in MovieExec, is a key of that relation, as it must be. The meaning of either of these two foreign key declarations is that whenever a value appears in the presC# component of a Studio tuple, that value must also appear in the cert# component of some MovieExec tuple. The one exception is that, should a particular Studio tuple have NULL as the value of its presC# component, there is no requirement that NULL appear as the value of a cert# component (but note that cert# is a primary key and therefore cannot have NULL's anyway). □

## 7.1.2 Maintaining Referential Integrity

The schema designer may choose from among three alternatives to enforce a foreign-key constraint. We can learn the general idea by exploring Example 7.1, where it is required that a presC# value in relation Studio also be a cert# value in MovieExec. The following actions will be prevented by the DBMS (i.e., a run-time exception or error will be generated).

a) We try to insert a new Studio tuple whose presC# value is not NULL and is not the cert# component of any MovieExec tuple.

b) We try to update a Studio tuple to change the presC# component to a non-NULL value that is not the cert# component of any MovieExec tuple.

c) We try to delete a MovieExec tuple, and its cert# component, which is not NULL, appears as the presC# component of one or more Studio tuples.

d) We try to update a `MovieExec` tuple in a way that changes the `cert#` value, and the old `cert#` is the value of `presC#` of some movie studio.

For the first two modifications, where the change is to the relation where the foreign-key constraint is declared, there is no alternative; the system has to reject the violating modification. However, for changes to the referenced relation, of which the last two modifications are examples, the designer can choose among three options:

1. *The Default Policy: Reject Violating Modifications.* SQL has a default policy that any modification violating the referential integrity constraint is rejected.

2. *The Cascade Policy.* Under this policy, changes to the referenced attribute(s) are mimicked at the foreign key. For example, under the cascade policy, when we delete the `MovieExec` tuple for the president of a studio, then to maintain referential integrity the system will delete the referencing tuple(s) from `Studio`. If we update the `cert#` for some movie executive from $c_1$ to $c_2$, and there was some `Studio` tuple with $c_1$ as the value of its `presC#` component, then the system will also update this `presC#` component to have value $c_2$.

3. *The Set-Null Policy.* Here, when a modification to the referenced relation affects a foreign-key value, the latter is changed to `NULL`. For instance, if we delete from `MoveExec` the tuple for a president of a studio, the system would change the `presC#` value for that studio to `NULL`. If we updated that president's certificate number in `MovieExec`, we would again set `presC#` to `NULL` in `Studio`.

These options may be chosen for deletes and updates, independently, and they are stated with the declaration of the foreign key. We declare them with `ON DELETE` or `ON UPDATE` followed by our choice of `SET NULL` or `CASCADE`.

**Example 7.2:** Let us see how we might modify the declaration of

```
Studio(name, address, presC#)
```

in Example 7.1 to specify the handling of deletes and updates in the

```
MovieExec(name, address, cert#, netWorth)
```

relation. Figure 7.1 takes the first of the `CREATE TABLE` statements in that example and expands it with `ON DELETE` and `ON UPDATE` clauses. Line (5) says that when we delete a `MovieExec` tuple, we set the `presC#` of any studio of which he or she was the president to `NULL`. Line (6) says that if we update the `cert#` component of a `MovieExec` tuple, then any tuples in `Studio` with the same value in the `presC#` component are changed similarly.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Figure 7.1: Choosing policies to preserve referential integrity

### Dangling Tuples and Modification Policies

A tuple with a foreign key value that does not appear in the referenced relation is said to be a *dangling tuple*. Recall that a tuple which fails to participate in a join is also called "dangling." The two ideas are closely related. If a tuple's foreign-key value is missing from the referenced relation, then the tuple will not participate in a join of its relation with the referenced relation, if the join is on equality of the foreign key and the key it references (called a *foreign-key join*). The dangling tuples are exactly the tuples that violate referential integrity for this foreign-key constraint.

Note that in this example, the set-null policy makes more sense for deletes, while the cascade policy seems preferable for updates. We would expect that if, for instance, a studio president retires, the studio will exist with a "null" president for a while. However, an update to the certificate number of a studio president is most likely a clerical change. The person continues to exist and to be the president of the studio, so we would like the presC# attribute in Studio to follow the change. □

## 7.1.3 Deferred Checking of Constraints

Let us assume the situation of Example 7.1, where presC# in Studio is a foreign key referencing cert# of MovieExec. Arnold Schwarzenegger retires as Governor of California and decides to found a movie studio, called La Vista Studios, of which he will naturally be the president. If we execute the insertion:

```
INSERT INTO Studio
VALUES('La Vista', 'New York', 23456);
```

we are in trouble. The reason is that there is no tuple of MovieExec with certificate number 23456 (the presumed newly issued certificate for Arnold Schwarzenegger), so there is an obvious violation of the foreign-key constraint.

One possible fix is first to insert the tuple for La Vista without a president's certificate, as:

```
INSERT INTO Studio(name, address)
VALUES('La Vista', 'New York');
```

This change avoids the constraint violation, because the La-Vista tuple is inserted with NULL as the value of presC#, and NULL in a foreign key does not require that we check for the existence of any value in the referenced column. However, we must insert a tuple for Arnold Schwarzenegger into MovieExec, with his correct certificate number before we can apply an update statement such as

```
UPDATE Studio
SET presC# = 23456
WHERE name = 'La Vista';
```

If we do not fix MovieExec first, then this update statement will also violate the foreign-key constraint.

Of course, inserting Arnold Schwarzenegger and his certificate number into MovieExec before inserting La Vista into Studio will surely protect against a foreign-key violation in this case. However, there are cases of *circular constraints* that cannot be fixed by judiciously ordering the database modification steps we take.

**Example 7.3:** If movie executives were limited to studio presidents, then we might want to declare cert# to be a foreign key referencing Studio(presC#); we would first have to declare presC# to be UNIQUE, but that declaration makes sense if you assume a person cannot be the president of two studios at the same time.

Now, it is impossible to insert new studios with new presidents. We can't insert a tuple with a new value of presC# into Studio, because that tuple would violate the foreign-key constraint from presC# to MovieExec(cert#). We can't insert a tuple with a new value of cert# into MovieExec, because that would violate the foreign-key constraint from cert# to Studio(presC#). □

The problem of Example 7.3 can be solved as follows.

1. First, we must group the two insertions (one into Studio and the other into MovieExec) into a single transaction.

2. Then, we need a way to tell the DBMS not to check the constraints until after the whole transaction has finished its actions and is about to commit.

To inform the DBMS about point (2), the declaration of any constraint — key, foreign-key, or other constraint types we shall meet later in this chapter — may be followed by one of DEFERRABLE or NOT DEFERRABLE. The latter is the default, and means that every time a database modification statement is executed, the constraint is checked immediately afterwards, if the modification could violate the foreign-key constraint. However, if we declare a constraint to be DEFERRABLE, then we have the option of having it wait until a transaction is complete before checking the constraint.

We follow the keyword DEFERRABLE by either INITIALLY DEFERRED or INITIALLY IMMEDIATE. In the former case, checking will be deferred to just before each transaction commits. In the latter case, the check will be made immediately after each statement.

**Example 7.4:** Figure 7.2 shows the declaration of Studio modified to allow the checking of its foreign-key constraint to be deferred until the end of each transaction. We have also declared presC# to be UNIQUE, in order that it may be referenced by other relations' foreign-key constraints.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT UNIQUE
        REFERENCES MovieExec(cert#)
        DEFERRABLE INITIALLY DEFERRED
);
```

Figure 7.2: Making presC# unique and deferring the checking of its foreign-key constraint

If we made a similar declaration for the hypothetical foreign-key constraint from MovieExec(cert#) to Studio(presC#) mentioned in Example 7.3, then we could write transactions that inserted two tuples, one into each relation, and the two foreign-key constraints would not be checked until after both insertions had been done. Then, if we insert both a new studio and its new president, and use the same certificate number in each tuple, we would avoid violation of any constraint. □

There are two additional points about deferring constraints that we should bear in mind:

- Constraints of any type can be given names. We shall discuss how to do so in Section 7.3.1.
- If a constraint has a name, say MyConstraint, then we can change a deferrable constraint from immediate to deferred by the SQL statement

  ```
  SET CONSTRAINT MyConstraint DEFERRED;
  ```

  and we can reverse the process by replacing DEFERRED in the above to IMMEDIATE.

### 7.1.4 Exercises for Section 7.1

**Exercise 7.1.1:** Our running example movie database of Section 2.2.8 has keys defined for all its relations.

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare the following referential integrity constraints for the movie database as in Exercise 7.1.1.

a) The producer of a movie must be someone mentioned in `MovieExec`. Modifications to `MovieExec` that violate this constraint are rejected.

b) Repeat (a), but violations result in the `producerC#` in `Movie` being set to NULL.

c) Repeat (a), but violations result in the deletion or update of the offending `Movie` tuple.

d) A movie that appears in `StarsIn` must also appear in `Movie`. Handle violations by rejecting the modification.

e) A star appearing in `StarsIn` must also appear in `MovieStar`. Handle violations by deleting violating tuples.

! **Exercise 7.1.2:** We would like to declare the constraint that every movie in the relation `Movie` must appear with at least one star in `StarsIn`. Can we do so with a foreign-key constraint? Why or why not?

**Exercise 7.1.3:** Suggest suitable keys and foreign keys for the relations of the PC database:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1. Modify your SQL schema from Exercise 2.3.1 to include declarations of these keys.

**Exercise 7.1.4:** Suggest suitable keys for the relations of the battleships database

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3. Modify your SQL schema from Exercise 2.3.2 to include declarations of these keys.

**Exercise 7.1.5:** Write the following referential integrity constraints for the battleships database as in Exercise 7.1.4. Use your assumptions about keys from that exercise, and handle all violations by setting the referencing attribute value to `NULL`.

a) Every class mentioned in `Ships` must be mentioned in `Classes`.

b) Every battle mentioned in `Outcomes` must be mentioned in `Battles`.

c) Every ship mentioned in `Outcomes` must be mentioned in `Ships`.

## 7.2 Constraints on Attributes and Tuples

Within a SQL `CREATE TABLE` statement, we can declare two kinds of constraints:

1. A constraint on a single attribute.

2. A constraint on a tuple as a whole.

In Section 7.2.1 we shall introduce a simple type of constraint on an attribute's value: the constraint that the attribute not have a `NULL` value. Then in Section 7.2.2 we cover the principal form of constraints of type (1): *attribute-based* `CHECK` *constraints*. The second type, the tuple-based constraints, are covered in Section 7.2.3.

There are other, more general kinds of constraints that we shall meet in Sections 7.4 and 7.5. These constraints can be used to restrict changes to whole relations or even several relations, as well as to constrain the value of a single attribute or tuple.

### 7.2.1 Not-Null Constraints

One simple constraint to associate with an attribute is `NOT NULL`. The effect is to disallow tuples in which this attribute is `NULL`. The constraint is declared by the keywords `NOT NULL` following the declaration of the attribute in a `CREATE TABLE` statement.

**Example 7.5:** Suppose relation `Studio` required `presC#` not to be `NULL`, perhaps by changing line (4) of Fig. 7.1 to:

```
4)    presC# INT REFERENCES MovieExec(cert#) NOT NULL
```

This change has several consequences. For instance:

- We could not insert a tuple into `Studio` by specifying only the name and address, because the inserted tuple would have `NULL` in the `presC#` component.
- We could not use the set-null policy in situations like line (5) of Fig. 7.1, which tells the system to fix foreign-key violations by making `presC#` be `NULL`.

□

## 7.2.2 Attribute-Based CHECK Constraints

More complex constraints can be attached to an attribute declaration by the keyword `CHECK` and a parenthesized condition that must hold for every value of this attribute. In practice, an attribute-based `CHECK` constraint is likely to be a simple limit on values, such as an enumeration of legal values or an arithmetic inequality. However, in principle the condition can be anything that could follow `WHERE` in a SQL query. This condition may refer to the attribute being constrained, by using the name of that attribute in its expression. However, if the condition refers to any other relations or attributes of relations, then the relation must be introduced in the `FROM` clause of a subquery (even if the relation referred to is the one to which the checked attribute belongs).

An attribute-based `CHECK` constraint is checked whenever any tuple gets a new value for this attribute. The new value could be introduced by an update for the tuple, or it could be part of an inserted tuple. In the case of an update, the constraint is checked on the new value, not the old value. If the constraint is violated by the new value, then the modification is rejected.

It is important to understand that an attribute-based `CHECK` constraint is not checked if the database modification does not change the attribute with which the constraint is associated. This limitation can result in the constraint becoming violated, if other values involved in the constraint do change. First, let us consider a simple example of an attribute-based check. Then we shall see a constraint that involves a subquery, and also see the consequence of the fact that the constraint is only checked when its attribute is modified.

**Example 7.6:** Suppose we want to require that certificate numbers be at least six digits. We could modify line (4) of Fig. 7.1, a declaration of the schema for relation

```
Studio(name, address, presC#)
```

to be

```
4)    presC# INT REFERENCES MovieExec(cert#)
                 CHECK (presC# >= 100000)
```

For another example, the attribute `gender` of relation

```
MovieStar(name, address, gender, birthdate)
```

was declared in Fig. 2.8 to be of data type CHAR(1) — that is, a single character. However, we really expect that the only characters that will appear there are 'F' and 'M'. The following substitute for line (4) of Fig. 2.8 enforces the rule:

```
4) gender CHAR(1) CHECK (gender IN ('F', 'M')),
```

Note that the expression ('F' 'M') describes a one-component relation with two tuples. The constraint says that the value of any gender component must be in this set. □

**Example 7.7:** We might suppose that we could simulate a referential integrity constraint by an attribute-based CHECK constraint that requires the existence of the referred-to value. The following is an *erroneous* attempt to simulate the requirement that the presC# value in a

```
Studio(name, address, presC#)
```

tuple must appear in the cert# component of some

```
MovieExec(name, address, cert#, netWorth)
```

tuple. Suppose line (4) of Fig. 7.1 were replaced by

```
4)      presC# INT CHECK
            (presC# IN (SELECT cert# FROM MovieExec))
```

This statement is a legal attribute-based CHECK constraint, but let us look at its effect. Modifications to Studio that introduce a presC# that is not also a cert# of MovieExec will be rejected. That is almost what the similar foreign-key constraint would do, except that the attribute-based check will also reject a NULL value for presC# if there is no NULL value for cert#. But far more importantly, if we change the MovieExec relation, say by deleting the tuple for the president of a studio, this change is invisible to the above CHECK constraint. Thus, the deletion is permitted, even though the attribute-based CHECK constraint on presC# is now violated. □

### 7.2.3 Tuple-Based CHECK Constraints

To declare a constraint on the tuples of a single table $R$, we may add to the list of attributes and key or foreign-key declarations, in $R$'s CREATE TABLE statement, the keyword CHECK followed by a parenthesized condition. This condition can be anything that could appear in a WHERE clause. It is interpreted as a condition about a tuple in the table $R$, and the attributes of $R$ may be referred to by name in this expression. However, as for attribute-based CHECK constraints, the condition may also mention, in subqueries, other relations or other tuples of the same relation $R$.

**Limited Constraint Checking: Bug or Feature?**

One might wonder why attribute- and tuple-based checks are allowed to be violated if they refer to other relations or other tuples of the same relation. The reason is that such constraints can be implemented much more efficiently than more general constraints can. With attribute- or tuple-based checks, we only have to evaluate that constraint for the tuple(s) that are inserted or updated. On the other hand, assertions must be evaluated every time any one of the relations they mention is changed. The careful database designer will use attribute- and tuple-based checks only when there is no possibility that they will be violated, and will use another mechanism, such as assertions (Section 7.4) or triggers (Section 7.5) otherwise.

The condition of a tuple-based `CHECK` constraint is checked every time a tuple is inserted into $R$ and every time a tuple of $R$ is updated. The condition is evaluated for the new or updated tuple. If the condition is false for that tuple, then the constraint is violated and the insertion or update statement that caused the violation is rejected. However, if the condition mentions some other relation in a subquery, and a change to that relation causes the condition to become false for some tuple of $R$, the check does not inhibit this change. That is, like an attribute-based `CHECK`, a tuple-based `CHECK` is invisible to other relations. In fact, even a deletion from $R$ can cause the condition to become false, if $R$ is mentioned in a subquery.

On the other hand, if a tuple-based check does not have subqueries, then we can rely on its always holding. Here is an example of a tuple-based `CHECK` constraint that involves several attributes of one tuple, but no subqueries.

**Example 7.8:** Recall Example 2.3, where we declared the schema of table `MovieStar`. Figure 7.3 repeats the `CREATE TABLE` statement with the addition of a primary-key declaration and one other constraint, which is one of several possible "consistency conditions" that we might wish to check. This constraint says that if the star's gender is male, then his name must not begin with `'Ms.'`.

In line (2), `name` is declared the primary key for the relation. Then line (6) declares a constraint. The condition of this constraint is true for every female movie star and for every star whose name does not begin with `'Ms.'`. The only tuples for which it is *not* true are those where the gender is male and the name *does* begin with `'Ms.'`. Those are exactly the tuples we wish to exclude from `MovieStar`. □

```
CREATE TABLE MovieStar (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE,
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

Figure 7.3: A constraint on the table MovieStar

### Writing Constraints Correctly

Many constraints are like Example 7.8, where we want to forbid tuples that satisfy two or more conditions. The expression that should follow the check is the OR of the negations, or opposites, of each condition; this transformation is one of "DeMorgan's laws": the negation of the AND of terms is the OR of the negations of the same terms. Thus, in Example 7.8 the first condition was that the star is male, and we used `gender = 'F'` as a suitable negation (although perhaps `gender <> 'M'` would be the more normal way to phrase the negation). The second condition is that the name begins with 'Ms.', and for this negation we used the NOT LIKE comparison. This comparison negates the condition itself, which would be `name LIKE 'Ms.%'` in SQL.

### 7.2.4 Comparison of Tuple- and Attribute-Based Constraints

If a constraint on a tuple involves more than one attribute of that tuple, then it must be written as a tuple-based constraint. However, if the constraint involves only one attribute of the tuple, then it can be written as either a tuple- or attribute-based constraint. In either case, we do not count attributes mentioned in subqueries, so even a attribute-based constraint can mention other attributes of the same relation in subqueries.

When only one attribute of the tuple is involved (not counting subqueries), then the condition checked is the same, regardless of whether a tuple- or attribute-based constraint is written. However, the tuple-based constraint will be checked more frequently than the attribute-based constraint — whenever any attribute of the tuple changes, rather than only when the attribute mentioned in the constraint changes.

### 7.2.5 Exercises for Section 7.2

**Exercise 7.2.1:** Write the following constraints for attributes of the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

a) The year cannot be before 1915.

b) The length cannot be less than 60 nor more than 250.

c) The studio name can only be Disney, Fox, MGM, or Paramount.

**Exercise 7.2.2:** Write the following constraints on attributes from our example schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1.

a) The speed of a laptop must be at least 2.0.

b) The only types of printers are laser, ink-jet, and bubble-jet.

c) The only types of products are PC's, laptops, and printers.

! d) A model of a product must also be the model of a PC, a laptop, or a printer.

**Exercise 7.2.3:** Write the following constraints as tuple-based `CHECK` constraints on one of the relations of our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

If the constraint actually involves two relations, then you should put constraints in both relations so that whichever relation changes, the constraint will be checked on insertions and updates. Assume no deletions; it is not always possible to maintain tuple-based constraints in the face of deletions.

a) A star may not appear in a movie made before they were born.

! b) No two studios may have the same address.

! c) A name that appears in `MovieStar` must not also appear in `MovieExec`.

! d) A studio name that appears in `Studio` must also appear in at least one `Movies` tuple.

!! e) If a producer of a movie is also the president of a studio, then they must be the president of the studio that made the movie.

**Exercise 7.2.4:** Write the following as tuple-based CHECK constraints about our "PC" schema.

a) A PC with a processor speed less than 2.0 must not sell for more than $600.

b) A laptop with a screen size less than 15 inches must have at least a 40 gigabyte hard disk or sell for less than $1000.

**Exercise 7.2.5:** Write the following as tuple-based CHECK constraints about our "battleships" schema:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) No class of ships may have guns with larger than a 16-inch bore.

b) If a class of ships has more than 9 guns, then their bore must be no larger than 14 inches.

! c) No ship can be in battle before it is launched.

! **Exercise 7.2.6:** In Examples 7.6 and 7.8, we introduced constraints on the gender attribute ofMovieStar. What restrictions, if any, do each of these constraints enforce if the value of gender is NULL?

## 7.3 Modification of Constraints

It is possible to add, modify, or delete constraints at any time. The way to express such modifications depends on whether the constraint involved is associated with an attribute, a table, or (as in Section 7.4) a database schema.

### 7.3.1 Giving Names to Constraints

In order to modify or delete an existing constraint, it is necessary that the constraint have a name. To do so, we precede the constraint by the keyword CONSTRAINT and a name for the constraint.

**Example 7.9:** We could rewrite line (2) of Fig. 2.9 to name the constraint that says attribute name is a primary key, as

```
2)      name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY,
```

Similarly, we could name the attribute-based CHECK constraint that appeared in Example 7.6 by:

```
4) gender CHAR(1) CONSTRAINT NoAndro
                  CHECK (gender IN ('F', 'M')),
```

Finally, the following constraint:

```
6)     CONSTRAINT RightTitle
           CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

is a rewriting of the tuple-based CHECK constraint in line (6) of Fig. 7.3 to give that constraint a name. □

## 7.3.2 Altering Constraints on Tables

We mentioned in Section 7.1.3 that we can switch the checking of a constraint from immediate to deferred or vice-versa with a SET CONSTRAINT statement. Other changes to constraints are effected with an ALTER TABLE statement. We previously discussed some uses of the ALTER TABLE statement in Section 2.3.4, where we used it to add and delete attributes.

ALTER TABLE statements can affect constraints in several ways. You may drop a constraint with keyword DROP and the name of the constraint to be dropped. You may also add a constraint with the keyword ADD, followed by the constraint to be added. Note, however, that the added constraint must be of a kind that can be associated with tuples, such as tuple-based constraints, key, or foreign-key constraints. Also note that you cannot add a constraint to a table unless it holds at that time for every tuple in the table.

**Example 7.10:** Let us see how we would drop and add the constraints of Example 7.9 on relation MovieStar. The following sequence of three statements drops them:

```
ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;
ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;
ALTER TABLE MovieStar DROP CONSTRAINT RightTitle;
```

Should we wish to reinstate these constraints, we would alter the schema for relation MovieStar by adding the same constraints, for example:

```
ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey
    PRIMARY KEY (name);
ALTER TABLE MovieStar ADD CONSTRAINT NoAndro
    CHECK (gender IN ('F', 'M'));
ALTER TABLE MovieStar ADD CONSTRAINT RightTitle
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

**Name Your Constraints**

Remember, it is a good idea to give each of your constraints a name, even if you do not believe you will ever need to refer to it. Once the constraint is created without a name, it is too late to give it one later, should you wish to alter it. However, should you be faced with a situation of having to alter a nameless constraint, you will find that your DBMS probably has a way for you to query it for a list of all your constraints, and that it has given your unnamed constraint an internal name of its own, which you may use to refer to the constraint.

These constraints are now tuple-based, rather than attribute-based checks. We cannot bring them back as attribute-based constraints.

The name is optional for these reintroduced constraints. However, we cannot rely on SQL remembering the dropped constraints. Thus, when we add a former constraint we need to write the constraint again; we cannot refer to it by its former name. □

## 7.3.3 Exercises for Section 7.3

**Exercise 7.3.1:** Show how to alter your relation schemas for the movie example:

```
Movie(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

in the following ways.

a) Make `title` and `year` the key for `Movie`.

b) Require the referential integrity constraint that the producer of every movie appear in `MovieExec`.

c) Require that no movie length be less than 60 nor greater than 250.

! d) Require that no name appear as both a movie star and movie executive (this constraint need not be maintained in the face of deletions).

! e) Require that no two studios have the same address.

**Exercise 7.3.2:** Show how to alter the schemas of the "battleships" database:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

to have the following tuple-based constraints.

a) `Class` and `country` form a key for relation `Classes`.

b) Require the referential integrity constraint that every battle appearing in `Outcomes` also appears in `Battles`.

c) Require the referential integrity constraint that every ship appearing in `Outcomes` appears in `Ships`.

d) Require that no ship has more than 14 guns.

! e) Disallow a ship being in battle before it is launched.

## 7.4 Assertions

The most powerful forms of active elements in SQL are not associated with particular tuples or components of tuples. These elements, called "triggers" and "assertions," are part of the database schema, on a par with tables.

- An assertion is a boolean-valued SQL expression that must be true at all times.
- A trigger is a series of actions that are associated with certain events, such as insertions into a particular relation, and that are performed whenever these events arise.

Assertions are easier for the programmer to use, since they merely require the programmer to state what must be true. However, triggers are the feature DBMS's typically provide as general-purpose, active elements. The reason is that it is very hard to implement assertions efficiently. The DBMS must deduce whether any given database modification could affect the truth of an assertion. Triggers, on the other hand, tell exactly when the DBMS needs to deal with them.

### 7.4.1 Creating Assertions

The SQL standard proposes a simple form of *assertion* that allows us to enforce any condition (expression that can follow `WHERE`). Like other schema elements, we declare an assertion with a `CREATE` statement. The form of an assertion is:

```
CREATE ASSERTION <assertion-name> CHECK (<condition>)
```

The condition in an assertion must be true when the assertion is created and must remain true; any database modification that causes it to become false will be rejected.[1] Recall that the other types of CHECK constraints we have covered can be violated under certain conditions, if they involve subqueries.

## 7.4.2 Using Assertions

There is a difference between the way we write tuple-based CHECK constraints and the way we write assertions. Tuple-based checks can refer directly to the attributes of that relation in whose declaration they appear. An assertion has no such privilege. Any attributes referred to in the condition must be introduced in the assertion, typically by mentioning their relation in a select-from-where expression.

Since the condition must have a boolean value, it is necessary to combine results in some way to make a single true/false choice. For example, we might write the condition as an expression producing a relation, to which NOT EXISTS is applied; that is, the constraint is that this relation is always empty. Alternatively, we might apply an aggregation operator like SUM to a column of a relation and compare it to a constant. For instance, this way we could require that a sum always be less than some limiting value.

**Example 7.11 :** Suppose we wish to require that no one can become the president of a studio unless their net worth is at least \$10,000,000. We declare an assertion to the effect that the set of movie studios with presidents having a net worth less than \$10,000,000 is empty. This assertion involves the two relations

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

The assertion is shown in Fig. 7.4. □

```
CREATE ASSERTION RichPres CHECK
    (NOT EXISTS
         (SELECT Studio.name
          FROM Studio, MovieExec
          WHERE presC# = cert# AND netWorth < 10000000
         )
    );
```

Figure 7.4: Assertion guaranteeing rich studio presidents

[1]However, remember from Section 7.1.3 that it is possible to defer the checking of a constraint until just before its transaction commits. If we do so with an assertion, it may briefly become false until the end of a transaction.

**Example 7.12:** Here is another example of an assertion. It involves the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

and says the total length of all movies by a given studio shall not exceed 10,000 minutes.

```
CREATE ASSERTION SumLength CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movies GROUP BY studioName)
);
```

As this constraint involves only the relation `Movies`, it seemingly could have been expressed as a tuple-based `CHECK` constraint in the schema for `Movies` rather than as an assertion. That is, we could add to the definition of table `Movies` the tuple-based `CHECK` constraint

```
CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movies GROUP BY studioName));
```

Notice that in principle this condition applies to every tuple of table `Movies`. However, it does not mention any attributes of the tuple explicitly, and all the work is done in the subquery.

Also observe that if implemented as a tuple-based constraint, the check would not be made on deletion of a tuple from the relation `Movies`. In this example, that difference causes no harm, since if the constraint was satisfied before the deletion, then it is surely satisfied after the deletion. However, if the constraint were a lower bound on total length, rather than an upper bound as in this example, then we could find the constraint violated had we written it as a tuple-based check rather than an assertion. □

As a final point, it is possible to drop an assertion. The statement to do so follows the pattern for any database schema element:

`DROP ASSERTION` <assertion name>

### 7.4.3 Exercises for Section 7.4

**Exercise 7.4.1:** Write the following assertions. The database schema is from the "PC" example of Exercise 2.4.1:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) No manufacturer of PC's may also make laptops.

**Comparison of Constraints**

The following table lists the principal differences among attribute-based checks, tuple-based checks, and assertions.

| Type of Constraint | Where Declared | When Activated | Guaranteed to Hold? |
|---|---|---|---|
| Attribute-based CHECK | With attribute | On insertion to relation or attribute update | Not if subqueries |
| Tuple-based CHECK | Element of relation schema | On insertion to relation or tuple update | Not if subqueries |
| Assertion | Element of database schema | On any change to any mentioned relation | Yes |

b) A manufacturer of a PC must also make a laptop with at least as great a processor speed.

c) If a laptop has a larger main memory than a PC, then the laptop must also have a higher price than the PC.

d) If the relation Product mentions a model and its type, then this model must appear in the relation appropriate to that type.

**Exercise 7.4.2:** Write the following as assertions. The database schema is from the battleships example of Exercise 2.4.3.

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) No class may have more than 2 ships.

! b) No country may have both battleships and battlecruisers.

! c) No ship with more than 9 guns may be in a battle with a ship having fewer than 9 guns that was sunk.

! d) No ship may be launched before the ship that bears the name of the first ship's class.

! e) For every class, there is a ship with the name of that class.

! **Exercise 7.4.3:** The assertion of Exercise 7.11 can be written as two tuple-based constraints. Show how to do so.

## 7.5 Triggers

*Triggers*, sometimes called *event-condition-action rules* or *ECA rules*, differ from the kinds of constraints discussed previously in three ways.

1. Triggers are only awakened when certain *events*, specified by the database programmer, occur. The sorts of events allowed are usually insert, delete, or update to a particular relation. Another kind of event allowed in many SQL systems is a transaction end.

2. Once awakened by its triggering event, the trigger tests a *condition*. If the condition does not hold, then nothing else associated with the trigger happens in response to this event.

3. If the condition of the trigger is satisfied, the *action* associated with the trigger is performed by the DBMS. A possible action is to modify the effects of the event in some way, even aborting the transaction of which the event is part. However, the action could be any sequence of database operations, including operations not connected in any way to the triggering event.

### 7.5.1 Triggers in SQL

The SQL trigger statement gives the user a number of different options in the event, condition, and action parts. Here are the principal features.

1. The check of the trigger's condition and the action of the trigger may be executed either on the *state of the database* (i.e., the current instances of all the relations) that exists before the triggering event is itself executed or on the state that exists after the triggering event is executed.

2. The condition and action can refer to both old and/or new values of tuples that were updated in the triggering event.

3. It is possible to define update events that are limited to a particular attribute or set of attributes.

4. The programmer has an option of specifying that the trigger executes either:

   (a) Once for each modified tuple (a *row-level trigger*), or

   (b) Once for all the tuples that are changed in one SQL statement (a *statement-level trigger*; remember that one SQL modification statement can affect many tuples).

Before giving the details of the syntax for triggers, let us consider an example that will illustrate the most important syntactic as well as semantic points. Notice in the example trigger, Fig. 7.5, the key elements and the order in which they appear:

a) The `CREATE TRIGGER` statement (line 1).

b) A clause indicating the triggering event and telling whether the trigger uses the database state before or after the triggering event (line 2).

c) A `REFERENCING` clause to allow the condition and action of the trigger to refer to the tuple being modified (lines 3 through 5). In the case of an update, such as this one, this clause allows us to give names to the tuple both before and after the change.

d) A clause telling whether the trigger executes once for each modified row or once for all the modifications made by one SQL statement (line 6).

e) The condition, which uses the keyword `WHEN` and a boolean expression (line 7).

f) The action, consisting of one or more SQL statements (lines 8 through 10).

Each of these elements has options, which we shall discuss after working through the example.

**Example 7.13:** In Fig. 7.13 is a SQL trigger that applies to the

```
MovieExec(name, address, cert#, netWorth)
```

table. It is triggered by updates to the `netWorth` attribute. The effect of this trigger is to foil any attempt to lower the net worth of a movie executive.

```
 1)  CREATE TRIGGER NetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD ROW AS OldTuple,
 5)      NEW ROW AS NewTuple
 6)  FOR EACH ROW
 7)  WHEN (OldTuple.netWorth > NewTuple.netWorth)
 8)      UPDATE MovieExec
 9)      SET netWorth = OldTuple.netWorth
10)      WHERE cert# = NewTuple.cert#;
```

Figure 7.5: A SQL trigger

Line (1) introduces the declaration with the keywords `CREATE TRIGGER` and the name of the trigger. Line (2) then gives the triggering event, namely the update of the `netWorth` attribute of the `MovieExec` relation. Lines (3) through (5) set up a way for the condition and action portions of this trigger to talk about both the old tuple (the tuple before the update) and the new tuple (the tuple after the update). These tuples will be referred to as `OldTuple` and `NewTuple`, according to the declarations in lines (4) and (5), respectively. In the condition and action, these names can be used as if they were tuple variables declared in the `FROM` clause of an ordinary SQL query.

Line (6), the phrase `FOR EACH ROW`, expresses the requirement that this trigger is executed once for each updated tuple. Line (7) is the condition part of the trigger. It says that we only perform the action when the new net worth is lower than the old net worth; i.e., the net worth of an executive has shrunk.

Lines (8) through (10) form the action portion. This action is an ordinary SQL update statement that has the effect of restoring the net worth of the executive to what it was before the update. Note that in principle, every tuple of `MovieExec` is considered for update, but the `WHERE`-clause of line (10) guarantees that only the updated tuple (the one with the proper `cert#`) will be affected. □

### 7.5.2 The Options for Trigger Design

Of course Example 7.13 illustrates only some of the features of SQL triggers. In the points that follow, we shall outline the options that are offered by triggers and how to express these options.

- Line (2) of Fig. 7.5 says that the condition test and action of the rule are executed on the database state that exists after the triggering event, as indicated by the keyword `AFTER`. We may replace `AFTER` by `BEFORE`, in which case the `WHEN` condition is tested on the database state that exists before the triggering event is executed. If the condition is true, then the action of the trigger is executed on that state. Finally, the event that awakened the trigger is executed, regardless of whether the condition is still true. There is a third option, `INSTEAD OF`, that we discuss in Section 8.2.3, in connection with modification of views.

- Besides `UPDATE`, other possible triggering events are `INSERT` and `DELETE`. The `OF netWorth` clause in line (2) of Fig. 7.5 is optional for `UPDATE` events, and if present defines the event to be only an update of the attribute(s) listed after the keyword `OF`. An `OF` clause is not permitted for `INSERT` or `DELETE` events; these events make sense for entire tuples only.

- The `WHEN` clause is optional. If it is missing, then the action is executed whenever the trigger is awakened. If present, then the action is executed only if the condition following `WHEN` is true.

- While we showed a single SQL statement as an action, there can be any number of such statements, separated by semicolons and surrounded by `BEGIN...END`.

- When the triggering event of a row-level trigger is an update, then there will be old and new tuples, which are the tuple before the update and after, respectively. We give these tuples names by the `OLD ROW AS` and `NEW ROW AS` clauses seen in lines (4) and (5). If the triggering event is an insertion, then we may use a `NEW ROW AS` clause to give a name for the inserted tuple, and `OLD ROW AS` is disallowed. Conversely, on a deletion `OLD ROW AS` is used to name the deleted tuple and `NEW ROW AS` is disallowed.

- If we omit the `FOR EACH ROW` on line (6) or replace it by the default `FOR EACH STATEMENT`, then a row-level trigger such as Fig. 7.5 becomes a statement-level trigger. A statement-level trigger is executed once whenever a statement of the appropriate type is executed, no matter how many rows — zero, one, or many — it actually affects. For instance, if we update an entire table with a SQL update statement, a statement-level update trigger would execute only once, while a row-level trigger would execute once for each tuple to which an update was applied.

- In a statement-level trigger, we cannot refer to old and new tuples directly, as we did in lines (4) and (5). However, any trigger — whether row- or statement-level — can refer to the relation of *old tuples* (deleted tuples or old versions of updated tuples) and the relation of *new tuples* (inserted tuples or new versions of updated tuples), using declarations such as `OLD TABLE AS OldStuff` and `NEW TABLE AS NewStuff`.

**Example 7.14:** Suppose we want to prevent the average net worth of movie executives from dropping below $500,000. This constraint could be violated by an insertion, a deletion, or an update to the `netWorth` column of

```
MovieExec(name, address, cert#, netWorth)
```

The subtle point is that we might, in one statement insert, delete, or change many tuples of `MovieExec`. During the modification, the average net worth might temporarily dip below $500,000 and then rise above it by the time all the modifications are made. We only want to reject the entire set of modifications if the net worth is below $500,000 at the end of the statement.

It is necessary to write one trigger for each of these three events: insert, delete, and update of relation `MovieExec`. Figure 7.6 shows the trigger for the update event. The triggers for the insertion and deletion of tuples are similar.

Lines (3) through (5) declare that `NewStuff` and `OldStuff` are the names of relations containing the new tuples and old tuples that are involved in the database operation that awakened our trigger. Note that one database statement can modify many tuples of a relation, and if such a statement executes, there can be many tuples in `NewStuff` and `OldStuff`.

```
 1)  CREATE TRIGGER AvgNetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD TABLE AS OldStuff,
 5)      NEW TABLE AS NewStuff
 6)  FOR EACH STATEMENT
 7)  WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
 8)  BEGIN
 9)      DELETE FROM MovieExec
10)      WHERE (name, address, cert#, netWorth) IN NewStuff;
11)      INSERT INTO MovieExec
12)          (SELECT * FROM OldStuff);
13)  END;
```

Figure 7.6: Constraining the average net worth

If the operation is an update, then tables NewStuff and OldStuff are the new and old versions of the updated tuples, respectively. If an analogous trigger were written for deletions, then the deleted tuples would be in OldStuff, and there would be no declaration of a relation name like NewStuff for NEW TABLE in this trigger. Likewise, in the analogous trigger for insertions, the new tuples would be in NewStuff, and there would be no declaration of OldStuff.

Line (6) tells us that this trigger is executed once for a statement, regardless of how many tuples are modified. Line (7) is the condition. This condition is satisfied if the average net worth *after* the update is less than $500,000.

The action of lines (8) through (13) consists of two statements that restore the old relation MovieExec if the condition of the WHEN clause is satisfied; i.e., the new average net worth is too low. Lines (9) and (10) remove all the new tuples, i.e., the updated versions of the tuples, while lines (11) and (12) restore the tuples as they were before the update. □

**Example 7.15:** An important use of BEFORE triggers is to fix up the inserted tuples in some way before they are inserted. Suppose that we want to insert movie tuples into

```
Movies(title, year, length, genre, studioName, producerC#)
```

but sometimes, we will not know the year of the movie. Since year is part of the primary key, we cannot have NULL for this attribute. However, we could make sure that year is not NULL with a trigger and replace NULL by some suitable value, perhaps one that we compute in a complex way. In Fig. 7.7 is a trigger that takes the simple expedient of replacing NULL by 1915 (something that could be handled by a default value, but which will serve as an example).

Line (2) says that the condition and action execute before the insertion event. In the referencing-clause of lines (3) through (5), we define names for

```
1)  CREATE TRIGGER FixYearTrigger
2)  BEFORE INSERT ON Movies
3)  REFERENCING
4)      NEW ROW AS NewRow
5)      NEW TABLE AS NewStuff
6)  FOR EACH ROW
7)  WHEN NewRow.year IS NULL
8)  UPDATE NewStuff SET year = 1915;
```

Figure 7.7: Fixing NULL's in inserted tuples

both the new row being inserted and a table consisting of only that row. Even though the trigger executes once for each inserted tuple [because line (6) declares this trigger to be row-level], the condition of line (7) needs to be able to refer to an attribute of the inserted row, while the action of line (8) needs to refer to a table in order to describe an update. □

### 7.5.3 Exercises for Section 7.5

**Exercise 7.5.1:** Write the triggers analogous to Fig. 7.6 for the insertion and deletion events on MovieExec.

**Exercise 7.5.2:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the "PC" example of Exercise 2.4.1:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) When updating the price of a PC, check that there is no lower priced PC with the same speed.

b) When inserting a new printer, check that the model number exists in Product.

! c) When making any modification to the Laptop relation, check that the average price of laptops for each manufacturer is at least $1500.

! d) When updating the RAM or hard disk of any PC, check that the updated PC has at least 100 times as much hard disk as RAM.

! e) When inserting a new PC, laptop, or printer, make sure that the model number did not previously appear in any of PC, Laptop, or Printer.

**Exercise 7.5.3:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the battleships example of Exercise 2.4.3.

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) When a new class is inserted into `Classes`, also insert a ship with the name of that class and a `NULL` launch date.

b) When a new class is inserted with a displacement greater than 35,000 tons, allow the insertion, but change the displacement to 35,000.

! c) If a tuple is inserted into `Outcomes`, check that the ship and battle are listed in `Ships` and `Battles`, respectively, and if not, insert tuples into one or both of these relations, with `NULL` components where necessary.

! d) When there is an insertion into `Ships` or an update of the `class` attribute of `Ships`, check that no country has more than 20 ships.

!! e) Check, under all circumstances that could cause a violation, that no ship fought in a battle that was at a later date than another battle in which that ship was sunk.

! **Exercise 7.5.4:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The problems are based on our running movie example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

You may assume that the desired condition holds before any change to the database is attempted. Also, prefer to modify the database, even if it means inserting tuples with `NULL` or default values, rather than rejecting the attempted modification.

a) Assure that at all times, any star appearing in `StarsIn` also appears in `MovieStar`.

b) Assure that at all times every movie executive appears as either a studio president, a producer of a movie, or both.

c) Assure that every movie has at least one male and one female star.

d) Assure that the number of movies made by any studio in any year is no more than 100.

e) Assure that the average length of all movies made in any year is no more than 120.

## 7.6 Summary of Chapter 7

✦ *Referential-Integrity Constraints*: We can declare that a value appearing in some attribute or set of attributes must also appear in the corresponding attribute(s) of some tuple of the same or another relation. To do so, we use a `REFERENCES` or `FOREIGN KEY` declaration in the relation schema.

✦ *Attribute-Based Check Constraints*: We can place a constraint on the value of an attribute by adding the keyword `CHECK` and the condition to be checked after the declaration of that attribute in its relation schema.

✦ *Tuple-Based Check Constraints*: We can place a constraint on the tuples of a relation by adding the keyword `CHECK` and the condition to be checked to the declaration of the relation itself.

✦ *Modifying Constraints*: A tuple-based check can be added or deleted with an `ALTER` statement for the appropriate table.

✦ *Assertions*: We can declare an assertion as an element of a database schema. The declaration gives a condition to be checked. This condition may involve one or more relations of the database schema, and may involve the relation as a whole, e.g., with aggregation, as well as conditions about individual tuples.

✦ *Invoking the Checks*: Assertions are checked whenever there is a change to one of the relations involved. Attribute- and tuple-based checks are only checked when the attribute or relation to which they apply changes by insertion or update. Thus, the latter constraints can be violated if they have subqueries.

✦ *Triggers*: The SQL standard includes triggers that specify certain events (e.g., insertion, deletion, or update to a particular relation) that awaken them. Once awakened, a condition can be checked, and if true, a specified sequence of actions (SQL statements such as queries and database modifications) will be executed.

## 7.7 References for Chapter 7

References [5] and [4] survey all aspects of active elements in database systems. [1] discusses recent thinking regarding active elements in SQL-99 and future standards. References [2] and [3] discuss HiPAC, an early prototype system that offered active database elements.

1. R. J. Cochrane, H. Pirahesh, and N. Mattos, "Integrating triggers and declarative constraints in SQL database systems," *Intl. Conf. on Very Large Databases*, pp. 567–579, 1996.

2. U. Dayal et al., "The HiPAC project: combining active databases and timing constraints," *SIGMOD Record* **17**:1, pp. 51–70, 1988.

3. D. R. McCarthy and U. Dayal, "The architecture of an active database management system," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 215–224, 1989.

4. N. W. Paton and O. Diaz, "Active database systems," *Computing Surveys* **31**:1 (March, 1999), pp. 63–103.

5. J. Widom and S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.