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 deletion or update of the

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

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

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

Exercise 7.1.2: 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 7.1.3: 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**
the relation M
so with a fore

Exercise 7.1
battleships d
from that exe
value to NULL

a) Every b

b) Every s

c) Every c

7.2 Co

Within a SQL

1. A cons

2. A cons

In Section 7.
value: the co
tion 7.2.2 we
CHECK const
in Section 7.
There ar
Sections 7.4
whole relatio
single attribu

7.2.1 No

One simple c
disallow tupl
keywords NO
statement.

Example 7
haps by cha

4)

This change

! **Exercise 7.1.4:** 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 7.1.5: Write the following referential integrity constraints for the battleships database as in Exercise 7.1.3. Use your assumptions about keys from that exercise, and handle all violations by setting the referencing attribute value to `NULL`.

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

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

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

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:

```
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 length cannot be less than 30 nor more than 500.

b) The year cannot be before 1909.

c) The genre can only be drama, comedy, sciFi, or teen.

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

b) The speed of a laptop must be at least 2.2.

c) The only types of printers are laser and ink-jet.

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

! c) No two movie executives may have the same address.

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

!! 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,
```

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 `movieTitle`, `movieYear`, and `starName` the key for `StarsIn`.

b) Require the referential integrity constraint that the president of every studio appear in `MovieExec`.

c) Require that no movie length be less than 30 nor greater than 500.

! 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 movie executives 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) Require that no ship has more than 15 guns.

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

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

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

! 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>)
```

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 laptops may also make printers.

lves the rela-

oducerC#)

exceed 10,000

ioName)

gly could have
a for Movies
ition of table

ioName));

table Movies.
ly, and all the

int, the check
vies. In this
t was satisfied
However, if the
pper bound as
e written it as

ement to do so

schema is from

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) If a laptop has a larger hard disk than a PC, then the laptop must also have a higher price than the PC.

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

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

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 3 ships.

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

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

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

! e) No ship may be launched before the ship that bears the name of the first ship's 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
that will i
Notice in t
they appea

a) The

b) A cl
uses

c) A RE
refer
upda
both

d) A cl
or o

e) The
(line

f) The
10).

Each of th
the examp

Example

Mov

table. It i
trigger is

1)
2)
3)
4)
5)
6)
7)
8)
9)
10)

```
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 inserting a new laptop, check that the model number exists in `Product`.

b) When updating the price of a printer, check that there is no lower priced printer of the same type.

! c) 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`.

! d) When making any modification to the `PC` relation, check that the average price of PC's for each manufacturer is at least $500.

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

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 with a displacement less than 20,000 tons, allow the insertion, but change the displacement to 20,000.

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

! c) When there is an insertion into `Ships` or an update of the `class` attribute of `Ships`, check that no country has more than 30 ships.

! d) 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.

!! 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 the average length of all movies made in any year is no more than 150.

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

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

d) Assure
presider

e) Assure
more th

7.6 Su

✦ *Referen*
in some
ing attr
we use

✦ *Attribu*
value o
be chec

✦ *Tuple-I*
of a rel
to the

✦ *Modify*
an ALT

✦ *Asserti*
schema
may in
the rel
individ

✦ *Invokir*
to one
only ch
by ins
they h

✦ *Trigge*
(e.g., i
them.
ified s
modifi

7.7 Re

References [
[1] discusses

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

e) Assure that the number of movies made by any studio in any year is no more than 50.

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