CHAPTER 7. CONSTRAINTS AND TRIGGERS

4. SCHEMA-LEVEL CONSTRAINTS AND TRIGGERS

7.4. SCHEMA-LEVEL CONSTRAINTS AND TRIGGERS

7.4.1 Assertions

The SQL standard proposes a simple form of assertion (also called a "general constraint") 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 <name> CHECK (<condition>);
```

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

There is a difference between the way we write tuple-based CHECK constraints and the way we write assertions. Tuple-based checks can refer to the attributes of that relation in whose declaration they appear. For instance, in line (6) of Fig. 7.5 we used attributes gender and name without saying where they came from. They refer to components of a tuple being inserted or updated in the table MovieStar, because that table is the one being declared in the CREATE TABLE statement.

The condition of 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 normal to aggregate the results of the condition 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 aggregate 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 7.13**: 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

```
MovieStar(name, address, cert#, netWorth)
Studio(name, address, presC#)
```
CREATE ASSERTION RichPres CHECK
    (NOT EXISTS
        (SELECT * FROM Studio, MovieExec
            WHERE pres$ = cert$ AND netWorth < 100000000)
    );

Figure 7.6: Assertion guaranteeing rich studio presidents

The assertion is shown in Fig. 7.6. Incidentally, it is worth noting that even though this constraint involves two relations, we could write it as tuple-based CHECK constraints on the two relations rather than as a single assertion. For instance, we can add to the CREATE TABLE statement of Example 7.3 a constraint on Studio as shown in Fig. 7.7.

CREATE TABLE Studio(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    pres$ INT REFERENCES MovieExec(cert$),
    CHECK (pres$ NOT IN
        (SELECT cert$ FROM MovieExec
            WHERE netWorth < 100000000)
    );

Figure 7.7: A constraint on Studio mirroring an assertion

Note, however, that the constraint of Fig. 7.7 will only be checked when a change to its relation, Studio occurs. It would not catch a situation where the net worth of some studio president, as recorded in relation MovieExec, dropped below $10,000,000. To get the full effect of the assertion, we would have to add another constraint to the declaration of the table MovieExec, requiring that the net worth be at least $10,000,000 if that executive is the president of a studio.

Example 7.14: Here is another example of an assertion. It involves the relation

Movie(title, year, length, inColor, studioName, producerCW$)

7.4. SCHEMA-LEVEL CONSTRAINTS AND TRIGGERS

Comparison of Constraints

<table>
<thead>
<tr>
<th>Type of Constraint</th>
<th>Where Declared</th>
<th>When Activated</th>
<th>Guaranteed to Hold?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute-based CHECK</td>
<td>With attribute</td>
<td>On insertion to relation or attribute update</td>
<td>Not if subqueries</td>
</tr>
<tr>
<td>Tuple-based CHECK</td>
<td>Element of relation schema</td>
<td>On insertion to relation or tuple update</td>
<td>Not if subqueries</td>
</tr>
<tr>
<td>Assertion</td>
<td>Element of database schema</td>
<td>On any change to any mentioned relation</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

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 Movie GROUP BY studioName)
    );

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

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

Notice that in principle this condition applies to every tuple of table Movie. 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 Movie. 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. ☐
7.4. SCHEMA-LEVEL CONSTRAINTS AND TRIGGERS

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. In this example, the trigger executes once for each tuple that is updated.

Example 7.15: We shall write an 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. The trigger declaration appears in Fig. 7.8.

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 7.8: An 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. If this phrase is missing or if it is replaced by the default FOR EACH STATEMENT, then the triggering would occur once for an SQL statement, no matter how many triggering-event changes to tuples it made. We would not then declare alias for old and new rows, but we might use OLD TABLE and NEW TABLE, introduced below.

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
7.4. Schema-Level Constraints and Triggers

row- or statement-level — can refer to the relation of old tuples (deleted
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 7.16: 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 INSERT or UPDATE statement insert
or change many tuples of MovieExec, and 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 7.9 shows the trigger for the
update event. The triggers for the insertion and deletion of tuples are similar
but slightly simpler.

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 (AVG(netWorth) > (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 7.9: Constraining the average net worth

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 state-
ment can modify many tuples of a relation, and if such a statement executes,
there can be many tuples in NewStuff and OldStuff.

If the operation is an update, then 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.

7.4.4 Instead-Of Triggers

There is a useful feature of triggers that did not make the SQL-99 standard, but figured into the discussion of the standard and is supported by some commercial systems. This extension allows BEFORE or AFTER to be replaced by INSTEAD OF; the meaning is that when an event awakens a trigger, the action of the trigger is done instead of the event itself.

This capability offers little when the trigger is on a stored table, but it is very powerful when used on a view. The reason is that we cannot really modify a view (see Section 6.7.4). An instead-of trigger intercepts attempts to modify the view and in its place performs whatever action the database designer deems appropriate. The following is a typical example.

Example 7.17: Let us recall the definition of the view of all movies owned by Paramount:

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

from Example 6.45. As we discussed in Example 6.49, this view is updatable, but it has the unexpected flaw that when you insert a tuple into ParamountMovie, the system cannot deduce that the studioName attribute is surely Paramount, so that attribute is NULL in the inserted Movie tuple.

A better result can be obtained if we create an instead-of trigger on this view, as shown in Fig. 7.10. Much of the trigger is unsurprising. We see the keyword INSTEAD OF on line (2), establishing that an attempt to insert into ParamountMovie will never take place.

Rather, we see in lines (5) and (6) the action that replaces the attempted insertion. There is an insertion into Movie, and it specifies the three attributes that we know about. Attributes title and year come from the tuple we tried to insert into the view; we refer to these values by the tuple variable NewRow that was declared in line (3) to represent the tuple we are trying to insert. The

7.4 SCHEMA-LEVEL CONSTRAINTS AND TRIGGERS

1. CREATE TRIGGER Paramountinsert
2. INSTEAD OF INSERT ON ParamountMovie
3. REFERENCING NEW ROW AS NewRow
4. FOR EACH ROW
5. INSERT INTO Movie(title, year, studioName)
6. VALUES(NewRow.title, NewRow.year, 'Paramount');

Figure 7.10: Trigger to replace an insertion on a view by an insertion on the underlying base table

value of attribute studioName is the constant 'Paramount'. This value is not part of the inserted tuple. Rather, we assume it is the correct studio for the inserted movie, because the insertion came through the view ParamountMovie.

7.4.5 Exercises for Section 7.4

Exercise 7.4.1: Write the triggers analogous to Fig. 7.9 for the insertion and deletion events on MovieExec.

Exercise 7.4.2: Write the following as triggers or assertions. 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 5.2.1:

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, 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) No manufacturer of PC's may also make laptops.

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

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

* e) When making any modification to the laptop relation, check that the average price of laptops for each manufacturer is at least $2000.

* f) 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.
7.5 SUMMARY OF CHAPTER 7

Movie(title, year, length, inColor, studioName, producercollision)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, prescollision)

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.

7.5 Summary of Chapter 7

Key Constraints: We can declare an attribute or set of attributes to be a
key with a UNIQUE or PRIMARY KEY declaration in a relation schema.

Referential Integrity Constraints: We can declare that a value appearing
in some attribute or set of attributes must also appear in the correspond-
ing attributes of some tuple of another relation with a REFERENCES or
FOREIGN KEY declaration in a 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.
CHAPTER 7. CONSTRAINTS AND TRIGGERS

 Assertions: We can declare an assertion as an element of a database schema with the keyword CHECK and the 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.

 Inactivating 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, these 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.

7.6 References for Chapter 7


Chapter 8

System Aspects of SQL

We now turn to the question of how SQL fits into a complete programming environment. In Section 8.1 we see how to embed SQL in programs that are written in an ordinary programming language, such as C. A critical issue is how we move data between SQL relations and the variables of the surrounding, or “host,” language.

Section 8.2 considers another way to combine SQL with general-purpose programming: persistent stored modules, which are pieces of code stored as part of a database schema and executable on command from the user. Section 8.3 covers additional system issues, such as support for a client-server model of computing.

A third programming approach is a “call-level interface,” where we program in some conventional language and use a library of functions to access the database. In Section 8.4 we discuss the SQL-standard library called SQL/CLI for making calls from C programs. Then, in Section 8.5 we meet Java’s JDBC (database connectivity), which is an alternative call-level interface.

Then, Section 8.6 introduces us to the “transaction,” an atomic unit of work. Many database applications, such as banking, require that operations on the data appear atomic, or indivisible, even though a large number of concurrent operations may be in progress at once. SQL provides features to allow us to specify transactions, and SQL systems have mechanisms to make sure that what we call a transaction is indeed executed atomically. Finally, Section 8.7 discusses how SQL controls unauthorized access to data, and how we can tell the SQL system what accesses are authorized.

8.1 SQL in a Programming Environment

To this point, we have used the generic SQL interface in our examples. That is, we have assumed there is an SQL interpreter, which accepts and executes the sorts of SQL queries and commands that we have learned. Although provided as an option by almost all DBMS’s, this mode of operation is actually rare. In