Data Modelling and Databases - Chapter 7 (Book)

- Author: Ruben Schenk
- Date: 10.05.2021
- Contact: ruben.schenk@inf.ethz.ch

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 ot execute at appropriate times.

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.

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 cosntraint is an assertion that values for certain attributes must make sense. In SQL we may decalre an attribute or attributes of one relation to be a `foreign key`, referencing some attribute(s) of a second relation. The implication of this declaration is twofold:

1. The referenced attribute(s) of the second realtion must be declared UNIQUE or the PRIMARY KEY for this relation.
2. Values of the foreign key appearing in the first relation must also appear in the referenced attributes of some tuple.

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

1. If the foreign key is a single attribute we may follow its name and type by a declaration that it *references" some attribute of some talbe. The form of the declaration is

```
    REFERENCES <table>(<attribute>)
```

2. 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. The form of this declaration is:

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

Example: Suppose we wish to declare the relation `Studio(name, address, presCNum)` whose primary key is *name* and which has a foreign key *presCNum* that references *certNum* of the relation `MovieExec(name, address, certNum, netWorth)`. We may declare *presCNum* directly to reference *certNum* as follows:

```
/* Code 7.1: Foreign keys in SQL. */
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCNum INT REFERENCES MovieExec(certNum)
);
```

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

```
/* Code 7.1: Continuation. */
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCNum INT,
    FOREIGN KEY (presCNum) REFERENCES MovieExec(certNum)
);
```

7.1.2 Maintaining Referential Integrity

Referring to example 7.1 from above, the DBMS will prevent the following actions:

- We try to insert a new *Studio* tuple whose *presCNum* value is not *NULL* and is not the *certNum* component of any *MovieExec* tuple.
- We try to update a *Studio* tuple to change the *presCNum* component to a non-NULL value that is not the *certNum* component of any *MovieExec* tuple.
- We try to delete a *MovieExec* tuple, and its *certNum* component, which ius not NULL, appears as the *presCNum* component of one or more *Studio* tuples.
- We try to update a *MovieExec* tuple in a way that changes the *certNum* value, and the old *certNum* is the value of *presCNum* 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.

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.
3. `The Set-Null Policy`. Here, when a modification to the referenced relation affects a foreign-key value, the latter is changed to NULL.

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

Example: We might modify our code example 7.1 the following way:

```
/* Code 7.2: Policy on changes to referenced relations. */
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCNum INT REFERENCES MovieExec(certNum)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

7.1.3 Deferred Checking of Constraints

Assume the situation of code example 7.1.Now, Arnold Schwarzenegger 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, so there is an abvious violation of the foreign-key constraint.
One possible fix is to first 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. 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 presCNum = 23456
WHERE name = 'La Vista';
```

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 make.

The problem shown in the example above can be solved as follows:

1. First, we must group the two insertions (one into *Studio* and the other into *MovieExec*) into a single transactions.
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 (e.g., key, foreign-key, etc.) may be followed by one of `DEFERRABLE` or `NOT DEFERRABLE`. The later is the default, and means that every time a database modification statement is executed, the constraint is checked immediately afterwards. 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.

7.2

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.

We will introduce the different types of constraints in the following subchapters.

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.

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 principle the condition can be anything that could follow WHERE in a SQL query.

An attribute-based `CHECK` constraint is checked *whenever any tuple gets a new value for this attribute*. The new value could be introduced by any update for the tuple, or it could be part of an inserted tuple.

Example: Suppose we want to require that certificate numbers be at least six digits. We could modify Code 7.1 to:

```
/* Code 7.6: Attribute CHECKS. */
presCNum INT REFERENCES MovieExec(certNum)
                CHECK (presCNum >= 100000)
```

7.2.3 Tuple-Base 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.

The condition of a tuple-base `CKECK` constraint is *checked every time a tuple is inserted into R and every time a tuple of R is updated*.
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 in R, the check does not inhibit this change. In fact, even a deletion from R can cause the condition to become false, if R is mentioned in a subquery.

Example: Assume we want to put a constraint on the tuples of the *MovieStar* relation, namely that if the star's gender is male, then his name must not begin with 'Ms.'. We can introduce this constraint the following way:

```
/* Code 7.8: Tuple CHECKS. */
CREATE TABLE MovieStar (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE,
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

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 cosntraint involves only one attribute of the tuple, then it can be written as either a tuple- or attribute- based constraint.

When only one attribute of the tuple is involved, 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 attrbiute-based constraint* - whenerver any attribute of the tuple changes, rather than only when the attribute mentioned in the constraint changes.

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 talbe, or 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 has a name. TO do so, we precede the constraint by the keyword `CONSTRAINT` and a name for the constraint.

Example: Assume we want to check that the gender attribute in the *MovieStar* relation is either *F* or *M*. We can introduce this check the following way:

```
/* Code 7.9: Giving names to constraints. */
gender CHAR(1) CONSTRAINT NoAndro CHECK (gender IN ('F', 'M'))
```

7.3.2 Altering Constraints on Tables

We mentioned in a previous section 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.

`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: Let us see how we would drop and add some constraints on the relation *MovieStar*. The following sequence of statements drops the *NoAndro* constraint and adds it again:

```
/* Code 7.10: Dropping and adding constraints to a relation. */
ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;

ALTER TABLE MovieStar ADD CONSTRAINT NoAndro
    CHECK (gender IN ('F', 'M'));
```

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 expresiion that must be true at all times.
- A `trigger` is a series of actions that are associated with certain events, such as insertion into a particular relation, and that are performed whenever events arise.

7.4.1 Creating Assertions

The SQL standard proposes a simple form of `assertions` that allos 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 modifiaction that causes it to become false will be rejected.

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.

Example: 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 dollars. We can declare an assertion to the effect that the set of movies studios with presidents having a net worth less than 10'000'000 dollars is empty. This assertion is shown in the following example code:

```
/* Code 7.11: Assertions in SQL. */
CREATE ASSERTION RichPres CHECK
    (NOT EXISTS
        (SELECT Studio.name
         FROM Studio, MovieExec
         WHERE presCNum = certNum AND netWorth < 10000000
        )
    );
```

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 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

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.
2. Once awakened by its triggering event, the trigger tests a `condition`. If the condition doesn't 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.

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* that exists before the triggering event itself is 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:
    1. Once for each modified tuple (a `row-level trigger`), or
    2. Once for all the tuples that are changed in one SQL statement (a `statement-level trigger`).

We now look at some details of the syntax for triggers with the following example code:

```
/* Code 7.13: A SQL trigger. */
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 certNum = NewTuple.certNum;
```

- Line (1) introduced 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*
- Lines (3) - (5) set up a way for the condition and action portions of the trigger to talk about both the old tuple and the new tuple
- Line (6) expresses the requirement that this trigger is executed once for each updated tuple
- Line (7) is the condition part of the trigger
- Lines (8) - (10) form the action portion

7.5.2 The Options for Trigger Design

In the points that follow, we shall outline the options that are offered by triggers and how to express these options.

- In line (2) 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.
- Besides UPDATE, other possible triggering events are INSERT and DELETE. The *OF netWorth* clause in line (2) is optional for UPDATE events and is not permitted for INSERT or DELETE events.
- The WHEN clause is optional. If it is missing, then the action is executed whenever the trigger is awakened.
- 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.
- If we omit the FOR EACH ROW on line (6) or replace it by the default FOR EACH STATEMENT, then a row-level trigger becomes a statement-level trigger. It is executed once whenever a statement of the appropriate type is executed, no matter how many rows it actually affects.
- In a statement-level trigger, we cannot refer to old and new tuples directly. Howevery, any trigger can refer to the relation of old tuples and the relation of new tuples, using declarations such as *OLD TABLE AS OldStuff* and *NEW TABLE AS NewStuff*.

7.6 Summary of Chapter 7

Referential-Integrity Constraints

WE can decalre 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 schema.

Attribute-Based Check Constraints

We can place a constraint on the value of any aatribute 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 decalre an `assertion` as an element of a database schema. The declaration gives a condition to be checked.

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.

Triggers

The SQL standard includes `triggers` that specify certain events that awaken them. Once awakened, a condition can be checked, and if true, a specified sequence of actions will be executed.