Chapter 7: Constraints and Triggers

Foreign Keys
Local and Global Constraints
Triggers
A **constraint** is a relationship among data elements that the DBMS is required to enforce.

- Example: key constraints.

**Triggers** are only executed when a specified condition occurs, e.g., insertion of a tuple.

- Easier to implement than complex constraints.
Kinds of Constraints

- **Keys.**
- **Foreign-key**, or referential-integrity.
- **Value-based** constraints.
  - Constrain values of a particular attribute.
- **Tuple-based** constraints.
  - Relationship among components.
- **Assertions**: any SQL boolean expression.
Foreign Keys

- Consider Relation `Sells(store, candy, price)`.
- We might expect that a candy value is a real candy --- something appearing in `Candies.name`.
- A constraint that requires a candy in `Sells` to be a candy in `Candies` is called a `foreign-key` constraint.
Expressing Foreign Keys

- Use the keyword REFERENCES, either:
  - Within the declaration of an attribute (only for one-attribute keys).
  - As an element of the schema:
    FOREIGN KEY ( <list of attributes> )
    REFERENCES <relation> ( <attributes> )
- Referenced attributes must be declared PRIMARY KEY or UNIQUE.
CREATE TABLE Candies (  
  name CHAR(20) PRIMARY KEY,  
  manf CHAR(20) );

CREATE TABLE Sells (  
  store CHAR(20),  
  candy CHAR(20) REFERENCES Candies(name),  
  price REAL );
CREATE TABLE Candies (  
    name CHAR(20) PRIMARY KEY,  
    manf CHAR(20) );

CREATE TABLE Sells (  
    store CHAR(20),  
    candy CHAR(20),  
    price REAL,  
    FOREIGN KEY(candy) REFERENCES Candies(name));
Enforcing Foreign-Key Constraints

- If there is a foreign-key constraint from attributes of relation $R$ to a key of relation $S$, two violations are possible:
  - An insert or update to $R$ introduces values not found in $S$.
  - A deletion or update to $S$ causes some tuples of $R$ to “dangle.”
Suppose $R = \text{Sells}$, $S = \text{Candies}$.

An insert or update to Sells that introduces a nonexistent candy must be rejected.

A deletion or update to Candies that removes a candy value found in some tuples of Sells can be handled in three ways (next slide).
Actions Taken --- (2)

- **Default**: Reject the modification.
- **Cascade**: Make the same changes in Sells.
  - Deleted candy: delete Sells tuple.
  - Updated candy: change value in Sells.
- **Set NULL**: Change the candy to NULL.
Example: Cascade

- Delete the Twizzler tuple from Candies:
  - Then delete all tuples from Sells that have candy = ’Twizzler’.
- Update the Twizzler tuple by changing ’Twizzler’ to ’Twiz.’:
  - Then change all Sells tuples with candy = ’Twizzler’ so that candy = ’Twiz.’.
Example: Set NULL

- Delete the Twizzler tuple from Candies:
  - Change all tuples of Sells that have candy = 'Twizzler' to have candy = NULL.
- Update the Twizzler tuple by changing 'Twizzler' to 'Twiz.':
  - Same change.
Choosing a Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.
- Follow the foreign-key declaration by:
  ON [UPDATE, DELETE][SET NULL, CASCADE]
- Two such clauses may be used.
- Otherwise, the default (reject) is used.
CREATE TABLE Sells ( 
    store CHAR(20),
    candy CHAR(20),
    price REAL,
    FOREIGN KEY (candy)
        REFERENCES Candies(name)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
Attribute-Based Checks

- Constraints on the value of a particular attribute.
- Add: CHECK( <condition> ) to the declaration for the attribute.
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.
Example

CREATE TABLE Sells (  
  store CHAR(20),
  candy CHAR(20) CHECK ( candy IN  
                        (SELECT name FROM Candies)),
  price REAL CHECK ( price <= 5.00 )  
);
Timing of Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated.

  **Example:** CHECK (price <= 5.00) checks every new price and rejects the modification (for that tuple) if the price is more than $5.

  **Example:** CHECK (candy IN (SELECT name FROM Candies)) is not checked if a candy is deleted from Candies (unlike foreign-keys).
Tuple-Based Checks

- CHECK ( <condition> ) may be added as a relation-schema element.
- The condition may refer to any attribute of the relation.
  - But any other attributes or relations require a subquery.
- Checked on insert or update only.
Example: Tuple-Based Check

- Only 7-11 can sell candy for more than $5:

```sql
CREATE TABLE Sells (
    store CHAR(20),
    candy CHAR(20),
    price REAL,
    CHECK (store = '7-11' OR price <= 5.00)
);
```
Assertions

- These are database-schema elements, like relations or views.
- Defined by:
  ```sql
  CREATE ASSERTION <name>
  CHECK ( <condition> );
  ```
- Condition may refer to any relation or attribute in the database schema.
Example: Assertion

In `Sells(store, candy, price)`, no store may charge an average of more than $5.

CREATE ASSERTION NoRipoffStores CHECK (NOT EXISTS (SELECT store FROM Sells GROUP BY stores HAVING AVG(price) > 5.00));

Stores with an average price above $5
Example: Assertion

In Consumers(name, addr, phone) and Stores (name, addr, license), there cannot be more stores than consumers.

CREATE ASSERTION FewStore CHECK ( 
   (SELECT COUNT(*) FROM Stores) <= 
   (SELECT COUNT(*) FROM Consumers) 
);
In principle, we must check every assertion after every modification to any relation of the database.

A clever system can observe that only certain changes could cause a given assertion to be violated.

**Example**: No change to Candies can affect FewStore. Neither can an insertion to Consumers.
Triggers: Motivation

- Assertions are powerful, but the DBMS often can’t tell when they need to be checked.
- Attribute- and tuple-based checks are checked at known times, but are not powerful.
- Triggers let the user decide when to check for a powerful condition.
Another name for “trigger” is ECA rule, or event-condition-action rule.

Event: typically a type of database modification, e.g., “insert on Sells.”

Condition: Any SQL boolean-valued expression.

Action: Any SQL statements.
Instead of using a foreign-key constraint and rejecting insertions into `Sells(store, candy, price)` with unknown candies, a trigger can add that candy to Candies, with a NULL manufacturer.
Example: Trigger Definition

CREATE TRIGGER CandyTrig
  AFTER INSERT ON Sells
  REFERENCING NEW ROW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.candy NOT IN
       (SELECT name FROM Candies))
  INSERT INTO Candies(name)
       VALUES(NewTuple.candy);
Example: Trigger Definition

CREATE TRIGGER CandyTrig
AFTER INSERT ON Sells
REFERENCING NEW ROW AS NewTuple
FOR EACH ROW
WHEN (NewTuple.candy NOT IN (SELECT name FROM Candies))
INSERT INTO Candies(name)
VALUES(NewTuple.candy);
Options: CREATE TRIGGER

- CREATE TRIGGER <name>
Options: The Event

- **AFTER can be BEFORE.**
  - Also, INSTEAD OF, if the relation is a view.
    - A great way to execute view modifications: have triggers translate them to appropriate modifications on the base tables.

- **INSERT can be DELETE or UPDATE.**
  - And UPDATE can be UPDATE . . . ON a particular attribute.
Options: FOR EACH ROW

- Triggers are either “row-level” or “statement-level.”
- FOR EACH ROW indicates row-level; its absence indicates statement-level.
- **Row level triggers**: execute once for each modified tuple.
- **Statement-level triggers**: execute once for an SQL statement, regardless of how many tuples are modified.
INSERT statements imply a new tuple (for row-level) or new table (for statement-level).
- The “table” is the set of inserted tuples.
DELETE implies an old tuple or table.
UPDATE implies both.
Refer to these by
[NEW OLD][ROW TABLE] AS <name>
Options: The Condition

- Any boolean-valued condition is appropriate.
- It is evaluated before or after the triggering event, depending on whether BEFORE or AFTER is used in the event.
- Access the new/old tuple or set of tuples through the names declared in the REFERENCING clause.
Options: The Action

- There can be more than one SQL statement in the action.
  - Surround by BEGIN . . . END if there is more than one.
- But queries make no sense in an action, so we are really limited to modifications.
Another Example

- Using \texttt{Sells(store, candy, price)} and a unary relation \texttt{RipoffStores(store)} created for the purpose, maintain a list of stores that raise the price of any candy by more than $1.
CREATE TRIGGER PriceTrig 
AFTER UPDATE OF price ON Sells 
REFERENCING 
   OLD ROW AS ooo 
   NEW ROW AS nnn 
FOR EACH ROW 
WHEN(nnn.price > ooo.price + 1.00) 
INSERT INTO RipoffStores 
   VALUES(nnn.store);
CREATE TRIGGER PriceTrig

AFTER UPDATE OF price ON Sells

REFERENCING
  OLD ROW AS ooo
  NEW ROW AS nnn

FOR EACH ROW

WHEN(nnn.price > ooo.price + 1.00)

INSERT INTO RipoffStores
VALUES(nnn.store);

The event – only changes to prices

Updates let us talk about old and new tuples

We need to consider each price change

Condition: a raise in price > $1

When the price change is great enough, add the store to RipoffStores
Triggers on Views

- Generally, it is impossible to modify a view, because it doesn’t exist.
- But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.
- Example: We’ll design a view Synergy that has (consumer, candy, store) triples such that the store sells the candy, the consumer frequents the store and likes the candy.
CREATE VIEW Synergy AS
  SELECT Likes.consumer, Likes.candy, Sells.store
  FROM Likes, Sells, Frequents
  WHERE Likes.consumer = Frequents.consumer
    AND Likes.candy = Sells.candy
    AND Sells.store = Frequents.store;
Example: The View

CREATE VIEW Synergy AS
SELECT Likes.consumer, Likes.candy, Sells.store
FROM Likes, Sells, Frequents
WHERE Likes.consumer = Frequents.consumer
    AND Likes.candy = Sells.candy
    AND Sells.store = Frequents.store;

Natural join of Likes, Sells, and Frequents
Interpreting a View Insertion

- We cannot insert into Synergy --- it is a view.
- But we can use an INSTEAD OF trigger to turn a \((\text{consumer}, \text{candy}, \text{store})\) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequent.
  - The Sells.price will have to be NULL.
CREATE TRIGGER ViewTrig
    INSTEAD OF INSERT ON Synergy
    REFERENCING NEW ROW AS n
    FOR EACH ROW
    BEGIN
        INSERT INTO LIKES VALUES(n.consumer, n.candy);
        INSERT INTO SELLS(store, candy) VALUES(n.store, n.candy);
        INSERT INTO FREQUENTS VALUES(n.consumer, n.store);
    END;