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 $\text{Sells}(\text{store, candy, price})$.
- We might expect that a candy value is a real candy --- something appearing in $\text{Candies.name}$.
- A constraint that requires a candy in $\text{Sells}$ to be a candy in $\text{Candies}$ is called a \textit{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.
Example: With Attribute

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.
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.
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 \texttt{Sells(store, candy, price)}, no store may charge an average of more than $5.

\begin{verbatim}
CREATE ASSERTION NoRipoffStores CHECK (NOT EXISTS (SELECT store FROM Sells
GROUP BY stores
HAVING AVG(price) > 5.00));
\end{verbatim}

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 \texttt{Sells(store, candy, price)} with unknown candies, a trigger can add that candy to \texttt{Candies}, with a \texttt{NULL} manufacturer.
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);
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.
Options: REFERENCING

- **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>
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 \textit{Sells}\textit{(store, candy, price)} and a unary relation \textit{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);
The Trigger

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
Condition: a raise in price > $1
We need to consider each price change
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 \((\text{consumer, candy, store})\) triples such that the store sells the candy, the consumer frequents the store, and the consumer likes the candy.
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;
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 (consumer, candy, store) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequentes.
  - The Sells.price will have to be NULL.
The Trigger

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;
Chapter 8: Views and Indexes
The simplest form of view definition is

```
CREATE VIEW <view-name> AS <view-definition>;
```

The view definition is a SQL query.

Example: CREATE VIEW MovieProducer AS

```
SELECT title, name
FROM Movies, MovieExec
WHERE producerC# = cert#;
```
Renaming Attributes

Example:

CREATE VIEW MovieProducer(movieTitle, prodName) AS
SELECT title, name
FROM Movies, MovieExec
WHERE producerC#, cert#
Example:

DROP VIEW MovieProducer
Indexes in SQL

Example:

SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year=1990;
Declaring Indexes

Example:

CREATE INDEX YearIndex ON Movies(year);

CREATE INDEX KeyIndex ON Movies(title, year);
Chapter 9

SQL In a Server Environment
Combining SQL and Conventional Programming Languages
Shortcomings of SQL

- Relational data model doesn't match well with data model of conventional programming languages (e.g., data structure mismatch)
- No pointers, loops or branches in SQL
- No convenient input and output (e.g., formatting)
We have seen only how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.

Reality is almost always different.

Programs in a conventional language like C are written to access a database by “calls” to SQL statements.
Three ways to combine:

- Persistent Stored Modules (code stored in the DB schema and executed on command from a user)
- Embed SQL statements in programs written in some ordinary language
- Call-level interfaces
  - SQL/CLI (SQL standard, for use with C)
  - JDBC (for use with Java)
Persistent Stored Modules

- A recent SQL standard
- Mechanism for user to store in the DB schema functions and procedures that can be used in SQL statements
- The functions and procedures are written in a simple general-purpose language
- Includes ifs, loops, variable declarations, as well as SQL queries and updates
- See Chapter 9 for more info.
Embedded SQL and CLI's

- host language + embedded SQL
  - preprocessor
  - host language + function calls (CLI)
    - host-language compiler
    - SQL library
    - object-code program
Host Languages

- Any conventional language can be a *host language*, that is, a language in which SQL calls are embedded.
- The use of a host/SQL combination allows us to do anything computable, yet still get the very-high-level SQL interface to the database.
Connecting SQL to the Host Language

- **Embedded SQL** is a standard for combining SQL with seven languages.
- **CLI (Call-Level Interface)** is a different approach to connecting C to an SQL database.
- **JDBC (Java Database Connectivity)** is a way to connect Java with an SQL database (analogous to CLI).
Embedded SQL

- **Key idea**: Use a preprocessor to turn SQL statements into procedure calls that fit with the host-language code surrounding.

- All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.
Issues for Embedded SQL

- how to transfer data between host language and SQL -- use shared variables
- how to handle multiple tuples returned by a query -- notion of a "cursor"
- how to execute SQL statements that are not known at compile time ("dynamic SQL")

See Chapter 9 for more details.
Instead of using a preprocessor, we can use a library of functions and call them as part of an ordinary C program.

- The library for C is called SQL/CLI = “Call-Level Interface.”
- Embedded SQL’s preprocessor will translate the EXEC SQL … statements into CLI or similar calls, anyway.
Java Database Connectivity (JDBC) is a library similar to SQL/CLI, but with Java as the host language.

JDBC/CLI differences are often related to the object-oriented style of Java, but there are other differences.
Overview of JDBC

- A "driver" for the database system to be used must be loaded. Result is creation of a DriverManager object.
- A connection object is obtained from the DriverManager in a somewhat implementation-dependent way.
- We’ll start by assuming we have myCon, a connection object.
JDBC provides two classes:

- **Statement** = an object that can accept a string that is an SQL statement and can execute such a string.
- **PreparedStatement** = an object that has an associated SQL statement ready to execute.
The Connection class has methods to create Statements and PreparedStatements.

Statement stat1 = myCon.createStatement();
PreparedStatement stat2 = myCon.createStatement(
"SELECT candy, price FROM Sells WHERE store = '7-11'");

Java trick: + concatenates strings.

createStatement with no argument returns a Statement; with one argument it returns a PreparedStatement.
Executing SQL Statements

- JDBC distinguishes queries from modifications, which it calls “updates.”
- Statement and PreparedStatement each have methods `executeQuery` and `executeUpdate`.
  - For Statements, these methods have one argument: the query or modification to be executed.
  - For PreparedStatements: no argument.
Example: Update

- stat1 is a Statement.
- We can use it to insert a tuple as:

```java
stat1.executeUpdate(
    "INSERT INTO Sells "+
    "VALUES('Safeway', 'Kitkat', 3.00)"
);
```
Example: Query

- stat2 is a PreparedStatement holding the query "SELECT candy, price FROM Sells WHERE store = '7-11' ".
- `executeQuery` returns an object of class ResultSet --- we’ll examine it later.
- The query:
  ```java
  ResultSet Menu = stat2.executeQuery();
  ```
Accessing the ResultSet

- An object of type ResultSet is something like a cursor (from PSM).
- Method `next()` advances the “cursor” to the next tuple.
  - The first time `next()` is applied, it gets the first tuple.
  - If there are no more tuples, `next()` returns the value FALSE.
When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.

Method $\text{getX}(i)$, where $X$ is some type, and $i$ is the component number, returns the value of that component.

- The value must have type $X$. 

Example: Accessing Components

- Menu is the ResultSet for the query "SELECT candy, price FROM Sells WHERE store = '7-11' ".
- Access the candy and price from each tuple by:

```java
while ( Menu.next() ) {
    theCandy = Menu.getString(1);
    thePrice = Menu.getFloat(2);
    /* do something with theCandy and thePrice */
}
```