Chapter 2

Constraints, Triggers, Views

Database System – The complete book,
Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom

Chapter Outline

- Keys and Foreign Key
- Constraints on Attributes and Tuples
- Modification of Constraints
- Trigger
- Views

Declaring Foreign-Key Constraints

- In SQL we may declare an attribute or attributes of one relation to be a foreign key, referencing some attribute(s) of a second relation (possibly the same relation)
- The referenced attribute(s) of the second relation must be declared UNIQUE or the PRIMARY KEY for their relation. Otherwise, we cannot make the foreign-key declaration.
- Values of the foreign key appearing in the first relation must also appear in the referenced attributes of some tuple.

Two ways to declare a foreign key.

- Example:

Suppose we wish to declare the relation

Studio(<u>name</u>, address, <u>presC#</u>)

which has a foreign key ***presC#*** that references ***cert#*** of relation:

MovieExec(name, addreses, cert#, netWorth)

We may declare presC# directly to reference cert# as follows:

```
CREATE TABLE Studio (
            name CHAR(30) PRIMARY KEY,
            address VARCHAR(255),
            presC# INT REFERENCES
                              MovieExec(cert#));
```

Two ways to declare a foreign key (cont.).

- Example:

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

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES
                                   MovieExec(cert#)
    );
```

Maintaining Referential Integrity.

- The following actions will be prevented by the DBMS (i.e., a run-time exception or error will be generated).

a) Insert a new Studio tuple whose presC# value is not NULL and is not the cert# component of any MovieExec tuple.

b) Update a Studio tuple to change the presC# component to a non-NULL value that is not the cert# component of any MovieExec tuple.

c) Delete a MovieExec tuple, and its cert# component, which is not NULL, appears as the presC# component of one or more Studio tuples.

d) Update a MovieExec tuple in a way that changes the cert# value, and the old cert# is the value of presC# of some movie studio.

Maintaining Referential Integrity (cont.)

- 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.
2. The Cascade Policy: Under this policy, changes to the referenced attribute(s) are mimicked at the foreign key.
3. The Set-Null Policy.

Maintaining Referential Integrity (cont.)

Example:

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presC# INT REFERENCES
                          MovieExec(cert#)
        ON DELETE SET NULL
        ON UPDATE CASCADE
        );
```

Constraints on Attributes and Tuples.

- **Not-Null Constraints:**

Example:

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
                                        NOT NULL
    );
```

- We could not insert a tuple into Studio by specifying only the name and address.
- We could not use the set-null policy

Constraints on Attributes and Tuples (cont.).

❑ **Attribute-Based CHECK Constraints :**

Example: Suppose we want to require that certificate numbers be at least six digits.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
                        CHECK (presC# >= 100000)
    );
```

- We could not insert a tuple into Studio by specifying only the name and address.
- We could not use the set-null policy

Constraints on Attributes and Tuples (cont.).

- **Tuple-Based CHECK Constraints :**

Example:

```
CREATE TABLE MovieStar (
       name CHAR(30) PRIMARY KEY,
       address VARCHAR(255),
       gender CHAR(1),
       birthdate DATE,
     CHECK (gender = ’F’ OR name NOT LIKE ’Ms.%’ )
       );
```

Modification of Constraints.

- **Giving Names to Constraints :**

Example:

```
CREATE TABLE MovieStar (
    name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE,
    CONSTRAINT RightTitle
          CHECK (gender = 'F' OR name NOT LIKE 'Ms.%' )
  );
```

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.

Modification of Constraints.

- **Altering Constraints on Tables:**

Example:

ALTER TABLE MovieStar **DROP CONSTRAINT** NameIsKey;

ALTER TABLE MovieStar **ADD CONSTRAINT** NameIsKey PRIMARY KEY (name);

Triggers.

- A trigger is a series of actions that are associated with certain events, and that are performed whenever these events arise.
- Triggers differ from the kinds of constraints in three points:

1. Triggers are only awakened when certain events, specified by the database programmer, occur (usually insert, delete, or update).
2. Once awakened by its triggering event, the trigger tests a condition.
3. If the condition of the trigger is satisfied, the action associated with the trigger is performed by the DBMS.

Triggers, sometimes called **event-condition-action rules** or **ECA rules**

Triggers.

- Example. MovieExec(name, address, cert#, netWorth)

```
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#;
```

The Options for Trigger Design

- We may replace AFTER by BEFORE (line 2) in which case the WHEN condition is tested on the database state that exists before the triggering event is executed.
- Besides UPDATE (line 2), other possible triggering events are INSERT and DELETE.
- The WHEN clause is optional. If it is missing, then the action is executed whenever the trigger is awakened.
- There can be any number of such statements (line 8-10), separated by semicolons and surrounded by BEGIN…END.

The Options for Trigger Design

- 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.
- A statement-level trigger is executed once whenever a statement of the appropriate type is executed, no matter how many rows — zero, one, or many — it actually affects.
- In a statement-level trigger, we cannot refer to old and new tuples directly (as line 4,5), but using declarations such as **OLD TABLE AS OldStuff** and **NEW TABLE AS NewStuff**.

The Options for Trigger Design

- Example

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING
        OLD TABLE AS OldStuff,
        NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN ((SELECT AVG(netWorth) FROM MovieExec) < 500000 )
BEGIN
    DELETE FROM MovieExec
        WHERE (name, address ,  cert#,  netWorth) IN NewStuff;
    INSERT INTO MovieExec
        (SELECT * FROM OldStuff ) ;
END;
```

Create trigger in SQL server

```
CREATE TRIGGER Trigger_name ON  {table | View}
  {FOR | AFTER | INSTEAD OF} { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
   AS { sql_statement [ ; ] [ ,...n ] [ ; ] > }
```

- Arguments

- **FOR | AFTER**: AFTER specifies that the DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires. AFTER is the default when FOR is the only keyword specified. AFTER triggers cannot be defined on views.

- INSTEAD OF: Specifies that the DML trigger is executed *instead of* the triggering SQL statement, therefore, overriding the actions of the triggering statements. INSTEAD OF cannot be specified for DDL or logon triggers. At most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view. INSTEAD OF triggers are not allowed on updatable views that use WITH CHECK OPTION.

- DML triggers are frequently used for enforcing business rules and data integrity.

Create trigger in SQL server

- Example 1

The following DML trigger prints a message to the client when anyone tries to add or change data in the Customertable in the AdventureWorks2012 database.

```
IF OBJECT_ID ('Sales.reminder1', 'TR') IS NOT NULL
  DROP TRIGGER Sales.reminder1;
GO
CREATE TRIGGER reminder1 ON Sales.Customer
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Customer Relations', 16, 10);
GO
```

Create trigger in SQL server

- Example 2

The following example sends an e-mail message to a specified person (MaryM) when the Customer table changes.

```
IF OBJECT_ID ('Sales.reminder2','TR') IS NOT NULL
DROP TRIGGER Sales.reminder2;
GO
CREATE TRIGGER reminder2 ON Sales.Customer
AFTER INSERT, UPDATE, DELETE
AS EXEC msdb.dbo.sp_send_dbmail
 @profile_name = 'AdventureWorks2012 Admin',
 @recipients = 'danw@Adventure-Works.com',
 @body = 'Don''t forget to print a report for the
          sales force.',
 @subject = 'Reminder';
GO
```

Create trigger in SQL server

- Example 3

```
CREATE TRIGGER NetWorthTrigger on MovieExec
AFTER UPDATE as
declare @new int, @old int, @ssn int
select @new=ne.netWorth, @old=ol.netWorth, @ssn=ol.name
from inserted ne, deleted ol
where ne.name=ol.name
if (@old > @new)
Begin
  UPDATE MovieExec
      SET netWorth = @old
      WHERE name = @ssn
end
GO
```

Views

- A *view* is a virtual table whose contents (columns and rows) are defined by a query which gets data in one or more tables (called *base tables*) or other views in the database.
- a view can be used for the following purposes:
  1. To focus, simplify, and customize the perception each user has of the database.
  2. As a security mechanism by allowing users to access data through the view, without granting the users permissions to directly access the underlying base tables.
  3. To provide a backward compatible interface to emulate a table whose schema has changed

Views

- Declare by:

```
CREATE VIEW view_name AS
     SELECT column_name(s)
     FROM table_name
     WHERE condition;
```

- Drop by:

```
  DROP VIEW <name>
```

- Modify by:

```
     ALTER VIEW view_name AS
     SELECT column_name(s)
     FROM table_name
     WHERE condition;
```

Our Running Example

Beers(Beername, manf)

Bars(Barname, addr, license)

Drinkers(Drname, addr, phone)

Likes(Drname, beer)

Sells(barname, beer, price)

Frequents(Drname, barname)

- Underline = *key* (tuples cannot have the same value in all key attributes).

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
   SELECT drinker, beer
   FROM Frequents, Sells
   WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

- Query a view as if it were a base table.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

```
SELECT beer FROM CanDrink
WHERE drinker = 'Sally';
```

Views

- Problem: each time a base table changes, the view may change.
  - **Note:** A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

Un-updatable Views

- Views defined using groups and aggregate functions are not updateable
- Views defined on multiple tables using joins are generally not updateable

Ex. Dept(did, budget, manageid)

Works(eid, did, pct_time)

```
create view cc as
 Select manageid,
     max(budget) as ln
 from Dept;
```

```
create view dd as
 Select d.did, w.eid,
         w.pct_time
 from Dept d, Works w
 where d.did = w.did;
```

Example

Movies (title, year, length, genre, studioName, producerC#)

A view is defined as:

```
CREATE VIEW ParamountMovies AS
        SELECT title , year
        FROM Movies
        WHERE studioName = ′Paramount′ ;
```

Suppose we insert into view ParamountMovies

```
        INSERT INTO ParamountMovies
                VALUES(′StarTrek′ , 1979);
```

What happen???

Triggers on Views

- An INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.

Example: Dept(did, budget, manageid)
Works(eid, did, pct_time)

```
create view dd as
 Select d.did, w.eid,
        w.pct_time
 from Dept d, Works w
 where d.did = w.did;
```

We can not directly insert into dd:

```
    INSERT INTO dd VALUES(9, 2, 16)
```

Interpreting a View Insertion

- We cannot insert into view dd.
- But we can use an INSTEAD OF trigger to turn a (did, eid, pct_time) triple into two insertions of projected pairs, one for each of Dept and Works.
  - Dept.budget and Dept.manageid will have to be NULL.

The Trigger on views

```
CREATE TRIGGER InsertToView ON  dd
INSTEAD OF INSERT
AS
DECLARE @sdid int, @seid int, @spct_time int
SELECT @sdid = inserted.did, @seid = inserted.eid,
          @spct_time = inserted.pct_time
FROM inserted
BEGIN
     INSERT INTO Dept(did) VALUES(@sdid)
     INSERT INTO Works VALUES(@seid, @sdid, @spct_time)
END
```

The Trigger in SQL-Server

```
CREATE TRIGGER GiveRaise1 ON Emp AFTER update
AS
Declare @new numeric(18,0), @old numeric(18,0), @eid smallint
SELECT @new=ne.Salary, @old=ol.salary, @eid = ne.eid
FROM Inserted ne, Deleted ol
where ne.eid = ol.eid
If @new>@old  BEGIN
      update Emp
      Set Emp.salary = @new
      where Emp.salary < @new and Emp.eid in
                                    ( Select D.manageid
                                       From Emp E, Works W, Dept D
                                       where E.eid = @eid
                                       and E.eid = W.eid
                                      and W.did = D.did); END
```

Q & A