Lecture 8
Constraints and Triggers

Instructor: Shel Finkelstein

Reference:
A First Course in Database Systems,
3rd edition, Chapter 7 (but not section 7.4)

Important Notices

- Lecture slides and other class information will be posted on Piazza (not Canvas).
  - Slides are posted under Resources→Lectures
  - Lecture Capture recordings are available to all students under Yuja.
    - That includes classes given over Zoom.
  - There's no Lecture Capture for Lab Sections.
- All Lab Sections (and Office Hours) normally meet In-Person, with rare exceptions announced on Piazza.
  - Some slides for Lab Sections have been posted on Piazza under Resources→Lab Section Notes
- Office Hours for TAs and me are posted in Syllabus and Lecture1, as well as in Piazza notice @7; that post also now includes hours for Group Tutors.
- Some suggestions about use (and non-use) of Generative AI systems such as ChatGPT were posted on Piazza in @41, with specific discussion about the Movies tables and the four Avatar queries in Lecture 4.
- Monday, February 19 is a holiday, Presidents day.
  - There are no Lectures, Lab Sections, Office Hours or Tutoring on that day.
  - Please attend a different Lab Section if you have a Monday Lab Section.

Important Notices

- CSE 180 Midterm was on **Wednesday, February 14**.
  - Midterm and Midterm answers were posted on Piazza under Resources→Exams on Wednesday, February 14.
  - Answers will be/were discussed at the beginning of class on Friday, February 16.
  - We'll finish grading and post grades as soon as possible.
- The Winter 2024 CSE 180 Final is on Wednesday, March 20 from 8:00am – 11:00am in our usual classroom.
- Third Gradiance Assignment, CSE 180 Winter 2024 #3, was assigned on Tuesday, February 6.
  - It was due on Thursday, February 15, by 11:59pm, the day after the Midterm.
  - There will be an announcement when we assign Gradiance #4, which will be after we complete much of Lecture 9, Relational Algebra.
- Lab2 grades were unmuted on Tuesday, February 13.
  - Please contact your TA and then me if you have questions about your grade by Monday, February 19.
  - Never contact the Reader who graded you Lab Assignment.

Important Notices

- Lab3 was posted on Piazza on Wednesday, February 14.
  - Lab3 is due on **Tuesday, February 27**.
    - Lab3 has many parts, and some parts of Lab3 are difficult.
  - We haven't discussed all of the Lab3 topics yet.
    - You'll have enough information to do Lab3 after we complete Lecture 8 (this lecture), probably on Friday, February 16.
    - You won’t need to use Triggers for Lab3.
  - You must use the Lab3 create file and load_lab3.sql file to do Lab3.
    - This original load data must be reloaded at multiple stages of Lab3.
    - The create_lab3.sql file was edited on Thursday, February 15 at about noon.
      - If you downloaded it before that, please download it again.

A Word to the Unwise

- This is a tough class for some students since it involves a combination of theory and practice.
  - The second half of CSE 180 is <u>much harder</u> than the first half of the course.
- Students who regularly attend Lectures and Lab Sections (and Office Hours and Tutoring) often do well; students who don't regularly attend often do poorly.
  - We don't take attendance; you're responsible for your own choices.
- After the course ends, your course grade will be determined by your scores on Exams, Labs and Gradiance, as described on the “Course Evaluation” and “Grading” Slides in the Syllabus.
  - You won't be able to do any additional work to improve your grade.

Constraints and Triggers

- A *constraint* is a relationship among data elements that the DBMS is required to enforce.
  - Example: key constraints.
- *Triggers/Rules* are only executed when a specified condition occurs, e.g., insertion of a tuple.
  - Easier to implement than complex constraints.

Kinds of Constraints

- Primary Key/Unique constraints
- Foreign Key, or referential-integrity constraints
- Attribute-based constraints
  - Constrain values of a particular attribute
- Tuple-based constraints
  - Relationship among components of tuple
- Assertions
  - Any SQL boolean expression (<u>not implemented</u> in most relational DBMS, not discussed in this lecture)

Review: Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

```
CREATE TABLE Beers (
    name  CHAR(20) UNIQUE,
    manf  CHAR(20)
);
```

Review: Multi-Attribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   VARCHAR(20),
  price  REAL,
  PRIMARY KEY (bar, beer)
);
```

Review: NULL

```
CREATE TABLE Sells (
      bar        CHAR(20),
      beer       VARCHAR(20),
      price REAL NOT NULL,
      PRIMARY KEY (bar, beer)
   );
```

If the CREATE statement didn't include NOT NULL for price:

```
ALTER TABLE Sells ALTER COLUMN price SET NOT NULL;

ALTER TABLE Sells ALTER COLUMN price DROP NOT NULL;
```

Foreign Keys

- Values appearing in attributes of one relation must also appear together in specific attributes of another relation.
- Example:
  - In Sells(bar, beer, price), we might expect that a beer value also appears in Beers.name (the name column of the Beers table, the primary key for that table).
- Like a link/pointer, but based on value.

CREATE TABLE for MovieExec

```
CREATE TABLE MovieExec (
    execName        CHAR(30),
    address         VARCHAR(255),
    cert#           INT PRIMARY KEY,
    netWorth        INTEGER
);
```

MovieExec(execName, address, <u>cert#</u>, netWorth)

CREATE TABLE for Studio

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

Studio(studioName, address, presC#)

- Every presC# in Studio must be the Primary Key (cert#) of a movie exec in the MovieExec table.
  - But for each movie exec in the MovieExec table, there can be n studios in the Studios table for which that movie exec is president, where n can be 0, 1, 2, …, 17, …

CREATE TABLE for Movies

```
CREATE TABLE Movies (
    movieTitle      CHAR(100),
    movieYear       INT,
    length          INT,
    genre           BOOLEAN,
    studioName      CHAR(30),
    producerC#      INT,
    PRIMARY KEY (movieTitle, movieYear),
    FOREIGN KEY (studioName) REFERENCES Studio,
    FOREIGN KEY (producerC#) REFERENCES MovieExec(cert#)
);
```

Didn't have to say:
FOREIGN KEY (studioName)
REFERENCES Studio (studioName),
because referencing attributes and referenced attributes are the same.

Movies(<u>movieTitle, movieYear</u>, length, genre, studioName, producerC#)

- Every studioName in Movies must be the Primary Key (studioName) of a studio in the Studio table.
  - But for each studioName in the Studio table, there can be n movies in the Movies table which have that studioName, where n can be 0, 1, 2, …, 17, …
- Every producerC# in Movies must be the Primary Key (cert#) of a movie exec in the MovieExec table.
  - But for each movie exec in the MovieExec table, there can be n movies in the Movies table for which that movie exec is producer, where n can be 0, 1, 2, …, 17, …

CREATE TABLE for MovieStar

```
CREATE TABLE MovieStar (
    starName        CHAR(30) PRIMARY KEY,
    address         VARCHAR(255),
    gender          CHAR(1),
    birthdate       DATE
);
```

MovieStar(starName, address, gender, birthdate)

CREATE TABLE for StarsIn

```
CREATE TABLE StarsIn (
    movieTitle CHAR(100),
    movieYear INT,
    starName        CHAR(30)
    PRIMARY KEY (movieTitle, movieYear, starName),
    FOREIGN KEY (movieTitle, movieYear) REFERENCES Movies,
    FOREIGN KEY (starName) REFERENCES MovieStar
);
```

StarsIn(<u>movieTitle, movieYear, starName</u>)

- Every (movieTitle, movieYear) in StarsIn must be the Primary Key of a movie in the Movies table.
  - But for each (movieTitle, movieYear) in the Movies table, there can be n Movies in the StarsIn table which have that (movieTitle, movieYear) , where n can be 0, 1, 2, …, 17, …

- Every starName in StarsIn must be the Primary Key of a movie star in the MovieStars table.
  - But for each starName in the MovieStar table, there can be n Movies in the StarsIn table which have that starName, where n can be 0, 1, 2, …, 17, …

Expressing Foreign Keys

- Use keyword REFERENCES, either:
  1. After an attribute (for one-attribute keys)
  2. As an element of the schema:

  FOREIGN KEY (<list of attributes>)

     REFERENCES <relation> [(<list of attributes>)]
- Referenced attributes must be declared as either PRIMARY KEY or UNIQUE.
  – (Why?)

Example: With Attribute

```
CREATE TABLE Beers (
    name    CHAR(20) PRIMARY KEY,
    manf    CHAR(20)
);

CREATE TABLE Sells (
    bar  CHAR(20),
    beer    CHAR(20) REFERENCES Beers(name),
    price   REAL,
    PRIMARY KEY(bar, beer)
);
```

Example: As a Schema Element

```
CREATE TABLE Beers (
    name    CHAR(20) PRIMARY KEY,
    manf    CHAR(20)
);

CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20),
    price   REAL,
    FOREIGN KEY(beer)
        REFERENCES Beers(name)
);
```

Example: As a Schema Element, With a Name for Constraint

```
CREATE TABLE Beers (
    name    CHAR(20) PRIMARY KEY,
    manf    CHAR(20)
);

CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20),
    price   REAL,
    CONSTRAINT BeerNameFK FOREIGN KEY(beer)
        REFERENCES Beers(name)
);
```

Adding/Dropping Foreign Key Constraint

```
CREATE TABLE Beers (
      name  CHAR(20) PRIMARY KEY,
      manf    CHAR(20)
);

CREATE TABLE Sells (
      bar      CHAR(20),
      beer     CHAR(20),
      price    REAL,
      PRIMARY KEY(bar, beer)
);

ALTER TABLE Sells
  ADD CONSTRAINT BeerNameFK FOREIGN KEY(beer)
        REFERENCES Beers(name);

ALTER TABLE Sells
  DROP CONSTRAINT BeerNameFK;
```

If we didn't include the BeerNameFK constraint in the CREATE statement …

… then we can add it, with a name, optionally…

… and any named constraint can also be dropped.

Enforcing Foreign Key Constraints (Referential Integrity, RI)

If there is a Foreign Key constraint from *referring relation R* (Child) to *referenced relation S (Parent),* then violations can occur in two ways:

1. An INSERT or UPDATE to *R* introduces values that are not found in *S*, or

2. A DELETE or UPDATE to S causes some tuples of *R* to "dangle", referencing a value that no longer exists

Actions Taken --- (1)

Example: Suppose *R* = Sells, *S* = Beers.

– That is, Sells refers to Beers

- An INSERT or UPDATE to Sells that introduces a non-existent beer must be rejected.

- A DELETE or UPDATE to Beers that removes a name that appears as a beer in some tuples of Sells can be handled in one of three ways (next slide).

Actions Taken --- (2)

1. ***RESTRICT***: Reject the modification. This is the Default.

2. ***CASCADE***: Make the same changes in Sells.
   - If DELETE of a beer in Beers that has Beers.name:
     Delete the Sells tuples whose Sells.beer attribute corresponds to Beers.name
   - If UPDATE of the name of a beer in Beers:
     Change the Sells.beer attribute of the tuples in Sells that corresponded to that old value of Beers.name …
     - … so that Sells.beer has the same new value as Beers.name

3. ***SET NULL***: Change Sells.beer to NULL because there's no longer a corresponding tuple in Beers with that Beers.name

Example: Cascade

- Upon DELETE of the Bud tuple from Beers:
  - Delete all tuples from Sells that have beer = ‘Bud'
- Upon UPDATE of the Bud tuple, changing ‘Bud' to ‘Budweiser':
  - Change all Sells tuples that have beer = ‘Bud' to have beer = ‘Budweiser'

Example: Set NULL

- Upon DELETE of the Bud tuple from Beers:
  - Change all tuples of Sells that have beer = 'Bud' so that their beer value becomes NULL.

- Upon UPDATE of the Bud tuple, changing 'Bud' to 'Budweiser':
  - Also change all tuples of Sells that have beer = 'Bud' so that their beer value becomes NULL.

- If Sells.beer can be NULL, then the relationship of Beers.name ("parent") to Sells.beer ("child") is **0/1:n**, not **1:n**.
  - That is, every "child" Foreign Key has a parent Primary Key, except for the children whose Foreign Key value is NULL.

  - Yes, you can set attribute values to NULL.

Choosing a Referential Integrity Policy

- When we declare a Foreign Key, we may choose policies SET NULL or CASCADE independently for DELETE and UPDATE, or stay with the Default (RESTRICT).

- Policy appears in the Foreign Key declaration with:

  ON [UPDATE, DELETE][SET NULL, CASCADE]

- Two such clauses may appear, one for UPDATE and one for DELETE

- If no policy is specified, then the Default (Reject) is used.
  - Specifying NO ACTION or RESTRICT explicitly specifies this default policy.
    - NO ACTION and RESTRICT are similar, but there is a subtle difference between them as to when constraint is checked.
  - Don't specify "Reject" as the policy.

Example: Setting Policy

```
CREATE TABLE Sells (
    bar      CHAR(20),
    beer     CHAR(20),
    price    REAL,
    PRIMARY KEY(bar, beer),
    FOREIGN KEY(beer)
        REFERENCES Beers(name)
            ON DELETE SET NULL
            ON UPDATE CASCADE
);
```

Attribute-Based Check

- Constraint on the value of a particular attribute.
- CHECK(<condition>) may be added to the declaration for the attribute.
  - Condition must evaluate to TRUE or UNKNOWN.
  - It's an Error if the check condition evaluates to FALSE.
- The condition may refer to the attribute of the relation that is being checked.
- For the condition to reference any other tuples or relations, a subquery must be used.
  - Note: Many Database Management Systems, including PostgreSQL, do not support CHECK with subquery.

Example: Attribute-Based Check

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20)    CHECK ( beer IN
                            (SELECT name FROM Beers) ),
    price   REAL        CHECK ( price <= 5.00 ),
    PRIMARY KEY (bar, beer)
);
```

Example: Named Check Constraints

```
CREATE TABLE Sells (
    bar    CHAR(20),
    beer CHAR(20)  CHECK ( beer IN
                                (SELECT name FROM Beers) ),
    price  REAL
    CONSTRAINT price_is_cheap
                        CHECK ( price <= 5.00 ),
    PRIMARY KEY (bar, beer)
);

ALTER TABLE Sells DROP CONSTRAINT price_is_cheap;
ALTER TABLE Sells ADD CONSTRAINT price_is_cheap
                                CHECK ( price <= 5.00 );
```

Timing of Attribute-Based 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 in Sells is more than $5.

- Example:
  CHECK (beer IN (SELECT name FROM Beers)) is not checked if a beer is deleted from Beers (unlike for Foreign Keys).

Tuple-Based Checks

- CHECK (<condition>) may be added as a relation-schema element.
- The condition may refer to any attributes of the relation (in the same tuple).
- But for the condition to reference any other tuples or relations, a subquery must be used.
  - Condition is checked **only** on INSERT or UPDATE into relation that has the CHECK.
  - Note: Many Database Management Systems, including PostgreSQL, do not support CHECK with subquery.

Example: Tuple-Based Check

Only Joe's Bar can sell beer for more than $5. That is:

IF price is more than 5.00 THEN bar must be Joe's Bar.

```
CREATE TABLE Sells (
  bar         CHAR(20),
  beer        CHAR(20),
  price       REAL,
  CHECK ( price <= 5.00  OR  bar = 'Joe''s Bar' ),
  PRIMARY KEY (bar, beer)
);
```

IF p THEN q is logically equivalent to: NOT p OR q

p is “price is more than 5.00” and q is “bar is Joe's Bar”.

When are Constraints Checked and Handled?

[This won't be on Exams]

- In the CONSTRAINT clause, you may specify a constraint to be DEFERRABLE or NOT DEFERRABLE
  - DEFERRABLE INITIALLY DEFERRED or
  - DEFERRABLE INITIALLY IMMEDIATE

- Within a transaction, SET CONSTRAINTS determines whether:
  - SET CONSTRAINTS { ALL | name [, ...] } IMMEDIATE;
    - Deferrable constraints are checked immediately for each SQL statement, or
  - SET CONSTRAINTS { ALL | name [, ...] } DEFERRED;
    - Checking is deferred until the end of the transaction.

Example: Deferred Constraint [This won't be on Exams]

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20)    CHECK ( beer IN
                        (SELECT name FROM Beers) ),
    price   REAL
    CONSTRAINT price_is_cheap
                          CHECK ( price <= 5.00 )
                DEFERRABLE INITIALLY DEFERRED,
    PRIMARY KEY (bar, beer)
);

SET CONSTRAINTS price_is_cheap IMMEDIATE;
```

Assertions

- These are database-schema elements, like relations or views.
- Defined by:

  CREATE ASSERTION <name>

      CHECK (<condition>);

- Condition may refer to any relation or attribute in the database schema.
- (**Not implemented** in Relational DBMS because they're too complicated and too expensive!)

Triggers: Motivation

- Assertions are powerful, it's difficult and expensive to implement them …
  - … so they're not implemented by DBMS.
- Attribute-based and tuple-based CHECKs are simpler and cheaper, but their power is very limited.
- **Triggers** let the user, perhaps a Database Administrator, (DBA), determine when to check for conditions and decide what to do when those conditions occur.

Event-Condition-Action Rules

- Another name for "trigger" is an *ECA Rule*, or **Event-Condition-Action** Rule.
  - **Event**: Typically a database modification, such as "INSERT ON Sells"
  - **Condition**: Any SQL boolean-valued expression
  - **Action**: Any SQL statements

Preliminary Example: A Trigger

- Instead of using a Foreign Key constraint and rejecting insertions into Sells(bar, beer, price) when the beer doesn't appear as a name in the relation
  Beers(name, manf)

- We can create a Trigger that adds a tuple to Beers with that beer as Beers.name, and with a NULL manufacturer.

Example: Trigger Definition

```
CREATE TRIGGER BeerTrig
  AFTER INSERT ON Sells
  REFERENCING NEW ROW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.beer NOT IN
      (SELECT name FROM Beers))
  INSERT INTO Beers(name)
      VALUES(NewTuple.beer);
```

The Event

The Condition

The Action

CREATE TRIGGER

- Either:

  CREATE TRIGGER <name>

- Or:

  CREATE OR REPLACE TRIGGER <name>

  – That's useful if there is (or might be) a Trigger with that name, and you want to modify that Trigger.

- CREATE OR REPLACE can also be used for other CREATE statements.

Options: The Event

- AFTER INSERT can be BEFORE INSERT.
  - Also, can use INSTEAD OF INSERT, e.g., if the relation is a view.
    - A clever way to execute modifications of a view is to have Triggers that translate them to appropriate modifications of the base tables.
- INSERT can be DELETE or UPDATE.
  - And UPDATE can be UPDATE OF a particular attribute.

Options: FOR EACH ROW

- Each trigger is either a “Row-level” trigger or a “Statement-level” trigger.
- FOR EACH ROW indicates that it's a Row-level trigger.
  - *Row-level trigger:*
    Trigger is executed once for each modified tuple.
- Not having FOR EACH ROW indicates that the trigger is a Statement-level trigger.
  - *Statement-level trigger:*
    Trigger is executed just once for the entire SQL statement, no matter how many tuples are modified by that statement.
  - Some systems don’t support Statement-level triggers.
    - We don’t show examples of Statement-level triggers.

Options: REFERENCING

- INSERT statement has a new tuple (for Row-level) or a new table (for Statement-level).
  - The "table" is the set of inserted tuples.
- DELETE statement has an old tuple or table.
- UPDATE statement has both old and new.
- In the Trigger, we can reference these by:

  [NEW OLD] [ROW TABLE] AS <name>

- Some systems support access to old row using keyword OLD, and to new row using keyword NEW, so REFERENCING clause isn't needed.
- [NEW OLD] TABLE is for Statement-level triggers.

Options: The Condition

- Any boolean-valued condition.

- Evaluated on the database as it existed before or after the Event, depending on whether BEFORE or AFTER is specified on the Event.
  - The Event may be INSERT, DELETE, UPDATE or SELECT.
  - Evaluation of the Condition always occurs <u>before</u> any triggered changes are executed, whether BEFORE Event or AFTER Event (or INSTEAD OF Event) is specified.

- Access the new/old row/table through the names in the REFERENCING clause.
  - Or just use NEW and OLD, in systems which support that.

Options: The Action

- There can be more than one SQL statement in the action.
  - Surround statement by BEGIN . . . END if there is more than one statement.

- Queries don't make sense in a triggered action, so statements in Trigger actions should be modification statements.
  - Some DBMS allow Trigger actions to be Stored Procedures, which we'll discuss later in the quarter.
  - PostgreSQL requires that Trigger action be a single Stored Procedure.

Another Example

- Using Sells(bar, beer, price) and a unary relation RipoffBars(bar), maintain a list of RipoffBars that raise the price of some beer by more than $1.

The Trigger

The Event: Only UPDATE of price.

```
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 RipoffBars
      VALUES(nnn.bar);
```

Updates let us talk about old and new tuples.

The Condition: price raise is more than $1.

We need to consider each price change

The Action: If the price change is big enough, add the bar to RipoffBars.