

Constraints and Triggers

Instructor: Shel Finkelstein

Reference:

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

Important Notices

- **Reminder:** Midterm is on **Monday, Nov 6** in usual classroom, at usual time.
 - **No make-ups**, no early/late exams.
 - You may bring a **single two-sided 8.5" x 11" sheet of paper** with as much info written (or printed) on it as you can fit and read unassisted.
 - **No sharing** of these sheets will be permitted.
 - Hand in these sheets at the end of the exam.
 - Midterm will cover all material up to and including **Lecture 7** (Views/Indexes).
 - **No devices** are permitted during exam.
 - You must show your **UCSC ID** at the end of the exam.
 - Hope that everyone who needs DRC accommodation has already submitted form to me.
 - Winter 2017 Midterm has been posted on Piazza; solution will be posted next week.
- Gradiance #3 is due on **Friday, November 3**; one problem uses Views!
- Sign-up for LSS tutoring with Alexander Ou ... if there's room.

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

- Keys/Unique constraints
- Foreign-key, or referential-integrity constraints
- Value-based constraints
 - Constrain values of a particular attribute
- Tuple-based constraints
 - Relationship among components of tuple
- Assertions
 - Any SQL boolean expression (not implemented 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**.

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> (<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 );
```

Example: As 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: Adding Foreign Key

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

Enforcing Foreign-Key Constraints (Referential Integrity, RI)

- If there is a foreign-key constraint from *referring relation* R to *referenced relation* S , then violations may occur two ways:
 1. An insert or update to R introduces values that are not found in S , or
 2. A deletion or update to S causes some tuples of R to “dangle”, referencing a value that no longer exists

Actions Taken --- (1)

- **Example:** suppose $R = \text{Sells}$, $S = \text{Beers}$.
 - That is, Sells refers to Beers
- An insert or update to Sells that introduces a nonexistent beer must be rejected.
- A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in one of three ways (next slide).

Actions Taken --- (2)

1. *Default* : Reject the modification.
2. *Cascade* : Make the same changes in Sells.
 - Deleted beer: delete Sells tuple.
 - Updated beer: also change value in Sells ...
 - ... so that Sells.beer has the same new value as Beers.name
3. *Set NULL* : Change Sells.beer to NULL.

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 by 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' to have beer = NULL.
- Upon Update of the Bud tuple, changing 'Bud' to 'Budweiser':
 - Make the same change to tuples of Sells that have beer='Bud' as for deletion (making beer=NULL).

Choosing a Referential Integrity 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 with:
`ON [UPDATE, DELETE][SET NULL, CASCADE]`
- Two such clauses may be used, one for UPDATE and one for DELETE
- Otherwise, the DEFAULT (Reject) is used.

Example: Setting Policy

```
CREATE TABLE Sells (  
    bar    CHAR(20),  
    beer   CHAR(20),  
    price  REAL,  
    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; can't be FALSE.
- The condition may refer to the attribute of the relation that is being checked.
- But for the condition to reference any other tuples or relations, a subquery must be used.
 - Note: PostgreSQL does not support CHECK with subquery. Sigh.

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 )  
);
```

Example: Named 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 )  
);
```

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

Timing of Attribute-Based Check

- 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 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 (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.

Example: Tuple-Based Check

- Only Joe's Bar can sell beer for more than \$5:

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

When are Constraints Checked and Handled?

- 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

```
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  
);  
  
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 most Relational DBMS because they're too complicated and expensive!)

Triggers: Motivation

- Assertions are powerful, but the DBMS often can't tell when they need to be checked ...
 - ... and they're probably not implemented by the DBMS.
- Attribute- and tuple-based checks are checked at known times, but they are not that powerful.
- Triggers let the user (often the DBA) decide when to check for any condition.

Event-Condition-Action Rules

- Another name for “trigger” is an *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

Preliminary **Example**: A Trigger

- Instead of using a foreign-key constraint and rejecting insertions into **Sells(bar, beer, price)** with unknown beers, a trigger can add that beer to Beers, 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>

- Useful if there is a trigger with that name and you want to modify the trigger.

Options: The Event

- AFTER INSERT can be BEFORE INSERT.
 - Also, can be INSTEAD OF, if the relation is a view.
 - A clever way to execute view modifications is to 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 a 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] [TUPLE TABLE] AS <name>

Options: The Condition

- Any boolean-valued condition.
- Evaluated on the database as it would exist before or after the triggering event, depending on whether BEFORE or AFTER is used.
 - But always before the changes take effect.
- Access the new/old tuple/table through the names 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 `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

