SQL: Modifications, Constraints & Triggers

CS 564- Fall 2016

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

Database Modifications

MODIFYING THE DB

- A modification command does not return a result, but it changes the database
- There are 3 kinds of modifications:
  1. **insert** tuple(s)
  2. **delete** tuple(s)
  3. **update** the value(s) of existing tuple(s)

INSERT

- To insert a single tuple:

  INSERT INTO *<relation>*

  VALUES *(<list of values>);*

- We may add to the relation name a list of attributes (if we forget the order)
- We may insert the entire output of a SQL query into a relation:

  INSERT INTO *<relation>*

  (<subquery>);

DELETE

- To delete tuples:

  DELETE FROM *<relation>*

  WHERE *<condition>;*

- How do we delete everything?

  DELETE FROM *<relation>;*

- Be careful: *all* tuples that satisfy the **WHERE** clause are deleted from the relation!

UPDATE

- To change certain attributes in certain tuples of a relation:

  UPDATE *<relation>*

  SET *<list of attribute assignments>*

  WHERE *<condition> ;*

- Example

```
UPDATE CountryLanguage
SET IsOfficial = 'T’
WHERE CountryCode = 'USA'
AND Language = 'Spanish';
```

VIEWS

VIEW DEFINITION

- A **view** is a virtual table, a relation that is defined in terms of the contents of other tables and views
- To create a view:

```
CREATE VIEW <name> AS <query> ;
```

- In contrast, a relation whose value is really stored in the database is called a base table

EXAMPLE

```
CREATE VIEW OfficialCountryLanguage AS
SELECT C.Name AS CountryName,
       L.Language AS Language
FROM CountryLanguage L, Country C
WHERE L.CountryCode = C.Code
     AND L.IsOfficial = 'T' ;
```

How To Use Views

- You may query a view as if it were a base table
- **BUT** there is a limited ability to modify views!
- The DBMS interprets the query as if the view were a base table
- The queries defining any views used by the query are replaced by their algebraic equivalents, and added to the expression tree for the query

CONSTRAINTS & TRIGGERS

CONSTRAINTS & TRIGGERS

- An **integrity constraint** is a relationship among data elements that the DBMS is required to enforce
  - Example: keys, foreign keys

- A **trigger** is a procedure that is executed when a specified condition occurs (e.g. tuple insertion)

Integrity Constraints (IC)

- keys (primary or unique)
- foreign-key
- domain constraints
  - e.g. NOT NULL
- tuple-based constraints
- assertions: any SQL boolean expression

KEYS

- To define a primary key:

```
CREATE TABLE Author(
    authorid INTEGER PRIMARY KEY,
    name TEXT) ;
```

- We can also define a unique key: a subset of attributes that uniquely defines a row (i.e. superkey):

```
CREATE TABLE Author(
    authorid INTEGER UNIQUE,
    name TEXT) ;
```

- There can be only one primary key, but many unique keys!

FOREIGN KEY

- Use the keyword **REFERENCES**, as:

FOREIGN KEY *(<list of attributes>)*
REFERENCES *<relation> (<attributes>)*

- Referenced attributes must be declared **PRIMARY KEY** or **UNIQUE**

Foreign Key

```
CREATE TABLE Author(
   authorid INTEGER PRIMARY KEY,
   name TEXT) ;

CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   author INTEGER,
   FOREIGN KEY (author) REFERENCES
   Author(authorid));
```

ENFORCING FOREIGN KEY CONSTRAINTS

If there is a foreign-key constraint from attributes of relation ***R*** to the primary key of relation ***S***, two violations are possible:

1. An insert or update to ***R*** introduces values not found in ***S***
2. A deletion or update to ***S*** causes some tuples of ***R*** to dangle

There are 3 ways to enforce foreign key constraints!

Action 1: Reject

- The insertion/deletion/update query is rejected and not executed in the DBMS
- This is the default action if a foreign key constraint is declared

Action 2: Cascade Update

- When a tuple referenced is *updated*, the update propagates to the tuples that reference it

```
CREATE TABLE Book(
    bookid INTEGER PRIMARY KEY,
    title TEXT,
    author INTEGER,
    FOREIGN KEY (author) REFERENCES
    Author(authorid)
    ON UPDATE CASCADE);
```

Action 2: Cascade Delete

- When a tuple referenced is *deleted*, the deletion propagates to the tuples that reference it

```
CREATE TABLE Book(
    bookid INTEGER PRIMARY KEY,
    title TEXT,
    author INTEGER,
    FOREIGN KEY (author) REFERENCES
    Author(authorid)
    ON DELETE CASCADE);
```

ACTION 3: SET NULL

- When a delete/update occurs, the values that reference the deleted tuple are set to **NULL**

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   author INTEGER,
   FOREIGN KEY (author) REFERENCES
   Author(authorid)
   ON UPDATE SET NULL);
```

WHAT SHOULD WE CHOOSE?

- When we declare a foreign key, we may choose policies **`SET NULL`** or **`CASCADE`** *independently* for deletions and updates

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

- Otherwise, the default policy (*reject*) is used

DOMAIN CONSTRAINTS

- A constraint on the value of a particular attribute:

  CHECK *(<condition>)*

```
CREATE TABLE Book(
    bookid INTEGER PRIMARY KEY CHECK(bookid >= 0),
    title TEXT,
    author INTEGER,
    FOREIGN KEY (author) REFERENCES Author(authorid));
```

DOMAIN CONSTRAINTS

- A check is **checked** only when a value for that attribute is inserted or updated
- We can also add more complex constraints:

```
CREATE TABLE Book(
    bookid INTEGER PRIMARY KEY,
    title TEXT,
    author INTEGER,
    FOREIGN KEY (author) REFERENCES Author(authorid)
    CHECK (bookid >= 0 OR title IS NOT NULL)
);
```

ASSERTIONS

- Defined by:

```
CREATE ASSERTION <name>
CHECK ( <condition> );
```

- The condition may refer to any relation or attribute in the database schema

```
CREATE ASSERTION LowPrice CHECK (
   NOT EXISTS (
      SELECT * FROM Book
      WHERE price <= 20 AND authorid = 111)
) ;
```

ASSERTIONS

- 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 and check only these

TRIGGERS: MOTIVATION

- Checks have limited capabilities
- Assertions are sufficiently general for most constraint applications, but they are hard to implement efficiently
- A trigger allows the user to specify when the check occurs

TRIGGERS

Procedure that starts automatically if specified changes occur to the DBMS

- Three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run)
  - **Action** (what happens if the trigger runs)

TRIGGER SYNTAX

```
CREATE TRIGGER <Trigger name>
{BEFORE | AFTER} {INSERT | DELETE | UPDATE}
      [OF <columns>] ON <Table name>

[REFERENCING {OLD | NEW} {ROW | TABLE}
   <reference name>]

[FOR EACH {ROW | STATEMENT}]
[WHEN (search condition)]
   SQL statement |
   BEGIN ATOMIC {SQL statements} END
```

EXAMPLE

```
CREATE TRIGGER addAuthor
AFTER INSERT ON Book
FOR EACH ROW
 WHEN (NEW.author NOT IN
   (SELECT authorid FROM Author))
BEGIN
INSERT INTO Author
   VALUES (NEW.author, 'NewAuthor') ;
END ;
```

TRIGGER: CONDITION

{BEFORE | AFTER}

- defines when the trigger action is executed relative to the trigger event

{INSERT | DELETE | UPDATE} ON

- defines the SQL modification that will activate the trigger
- in the case of *update*, we can specify the columns that when changed will activate the update

TRIGGER: REFERENCING

- **INSERT** statements imply a new tuple or new set of tuples
- **DELETE** implies an old tuple or table
- **UPDATE** implies both
- We can refer to these by using the **REFERENCING** clause to create aliases:

```
REFERENCING {NEW|OLD}{TUPLE|TABLE} <name>
```

TRIGGER: FOR EACH

Triggers are either row-level or statement-level

- **FOR EACH ROW** indicates row-level
- Row level triggers are executed once for each modified tuple
- **FOR EACH TABLE** (or absence of **FOR EACH**) indicates statement-level
- Statement-level triggers execute once for an SQL statement, regardless of how many tuples are modified

TRIGGER: ACTION

- There can be more than one SQL statement in the action clause
  - Surround by **`BEGIN ATOMIC . . . END`**

- SQL queries make no sense as an action, so we are essentially limited to modifications!

TRIGGERS VS CONSTRAINTS

- Both maintain data consistency
- Constraints are declarative, triggers are operational
- Triggers are more expressive, constraints are easier to understand
- Trigger use cases:
  - complex app actions (e.g., enforce credit limits)
  - auto-complete forms
  - generate logs

RECAP

- SQL modifications
  - **`INSERT, DELETE, UPDATE`**
- Views
- Integrity Constraints
  - primary/unique key
  - foreign key
  - domain constraints
  - assertions
- Triggers