Constraints and Triggers

Local and Global Constraints
Triggers

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.

Constraints and 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.

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.
Example: Assertion

- In `Sells(bar, beer, price)`, no bar may charge an average of more than $5.
- `CREATE ASSERTION NoRipoffBars CHECK (NOT EXISTS (SELECT bar FROM Sells GROUP BY bar HAVING 5.00 < AVG(price)))`};

Bars with an average price above $5

Example: Assertion

- In `Drinkers(name, addr, phone)` and `Bars(name, addr, license)`, there cannot be more bars than drinkers.
- `CREATE ASSERTION FewBar CHECK ((SELECT COUNT(*) FROM Bars) <= (SELECT COUNT(*) FROM Drinkers))`;

Timing of Assertion Checks

- 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 `Beers` can affect `FewBar`. Neither can an insertion to `Drinkers`.

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 any condition.
Event-Condition-Action Rules

◆ Another name for “trigger” is \textit{ECA rule}, or \textit{event-condition-action} rule.
◆ \textit{Event}: typically a type of database modification, e.g., “insert on Sells.”
◆ \textit{Condition}: Any SQL boolean-valued expression.
◆ \textit{Action}: Any SQL statements.

Preliminary Example: A Trigger

◆ Instead of using a foreign-key constraint and rejecting insertions into \texttt{Sells(bar, beer, price)} with unknown beers, a trigger can add that beer to \texttt{Beers}, with a NULL manufacturer.

Example: Trigger Definition

\begin{verbatim}
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);
\end{verbatim}

Options: CREATE TRIGGER

◆ \texttt{CREATE TRIGGER <name>}
◆ Or:
\texttt{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** can be **BEFORE**.
  - Also, **INSTEAD OF**, if the relation is a view.
    - A clever 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 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 bars that raise the price of any beer by more than $1.

The Trigger

```sql
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);
```

MySQL Triggers

Supplement MIS G2020
MySQL Trigger Basics

- Supported from release 5.0.2
  - Documentation only found at MySQL website
- Defined as “named database object that is associated with a table, and that activates when a particular event occurs for the table”
  - Activated with insert, update, or delete for the table attached
- Still remains very rudimentary!

Several Limitations

- Cannot create on view or temporary table
  - No INSTEAD OF trigger possible
- Requires SUPER privilege
  - SUPER can terminate other clients or change how the server operates
  - TRIGGER privilege from release 5.1.6
- Not activated by cascaded foreign key actions (soon available)
- Only row-level triggers

Trigger Creation

```
CREATE [ DEFINER = { user | CURRENT_USER } ]
TRIGGER trigger_name BEFORE | AFTER
INSERT | UPDATE | DELETE ON tbl_name
FOR EACH ROW trigger_stmt
```

- No two triggers for a given table that have the same trigger action time and event
- Trigger info is found at TRIGGERS table in INFORMATION_SCHEMA
- DROP [IF EXISTS] removes definition

Example I

```
CREATE TRIGGER testref
BEFORE INSERT ON test1 FOR EACH ROW
BEGIN
  INSERT INTO test2 SET a2 = NEW.a1;
  DELETE FROM test3
  WHERE a3 = NEW.a1;
  UPDATE test4 SET b4 = b4 + 1
  WHERE a4 = NEW.a1;
END;
```
NEW and OLD

- INSERT trigger has only NEW.col_name (no old row); DELETE trigger has only OLD.col_name (no new row); UPDATE trigger can have both
  - OLD is read-only
  - NEW can be referred to with SELECT privilege for it
  - BEFORE trigger can change with SET NEW.col_name = value with UPDATE privilege for it

Example II

```sql
CREATE TRIGGER upd_check
BEFORE UPDATE ON account FOR EACH ROW
BEGIN
  IF NEW.amount < 0 THEN SET NEW.amount = 0;
  ELSEIF NEW.amount > 100 THEN SET NEW.amount = 100;
  END IF;
END;
```

Trigger and Stored Routine

- Stored procedure (i.e., call statement) can be used
  - This is also advantageous if the same routine from within several triggers is invoked
- CALL statement cannot return data
  - Stored procedures are allowed to return data to the trigger through OUT or INOUT params
- Stored procedure cannot change the same table (risk of infinite calls)

Example III

```sql
CREATE TABLE account (
  acct_num INT,
  amount DECIMAL(10,2));

CREATE TRIGGER ins_sum
BEFORE INSERT ON account
FOR EACH ROW
SET @sum = @sum + NEW.amount;
```
Example III (cont.)

SET @sum = 0;
INSERT INTO account VALUES
    (137, 14.98), (141, 1937.50), (97, -100.00);
SELECT @sum AS 'Total amount';

+--------------+
| Total amount |
+--------------+
| 1852.48      |
+--------------+

User Defined Variables

- You can store a value in a user-defined variable and then refer to it later
  - For passing values from one statement to another
  - Variables are defined only within a client
  - All variables are freed when client exits
- User variables are written as \(@var\_name\)
  - \(var\_name\) may consist of alphanumeric characters from the current character set, ‘.’, ‘_’, and ‘$’

Error Handling

- Failing BEFORE trigger inactivates the operation on the corresponding row
- BEFORE trigger runs regardless of whether the post attempt subsequently succeeds
- AFTER trigger runs only if BEFORE trigger (if any) succeeds
- Error in BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation