Triggers and Events

Kathleen Durant PhD

Triggers

- Trigger: procedure that starts automatically if specified change occurs to the DBMS

- A trigger has three parts:
  - **Event**
    - Change to the database that activates the trigger
  - **Condition**
    - Query or test that is run when the trigger is activated
  - **Action**
    - Procedure that is executed when the trigger is activated and its ***Condition*** is true

Trigger Options

- **Event:** can be INSERT, DELETE, or UPDATE on DB table
- **Condition:**
  - Condition can be a true/false statement
    - All employee salaries are less than $100K
  - Condition can be a query
    - Interpreted as true if and only if answer set is not empty
- **Action:** can perform DB queries and updates that depend on:
  - Answers to query in condition part
  - Old and new values of tuples modified by the statement that activated the trigger
    - Old.field1 or New.field1
  - Action can also contain data-definition commands, e.g., create new tables

When to Fire the Trigger

- Triggers can be a row-level or a statement-level trigger
  - Row-level trigger: trigger executed once per modified record
  - Statement level trigger: executed once per activating statement

- Triggers can be executed before or after the activating SQL statement
  - Consider triggers on insertions
    - Trigger that initializes a variable for counting how many new tuples are inserted: execute **trigger before insertion**
    - Trigger that updates this count variable for each inserted tuple: **execute after each tuple is inserted** (might need to examine values of tuple to determine action)
    - Trigger can also be run **in place of the action**

MY SQL Trigger

CREATE TRIGGER <trigger-name> Trigger_time Trigger_event
ON table_name
FOR EACH ROW
BEGIN
END

- Syntax
  - Trigger_time is [BEFORE | AFTER]
  - Trigger_event [INSERT|UPDATE|DELETE]
  - Other key words – OLD AND NEW
  - Naming convention for a trigger trigger_time_tablename_trigger_event
  - Found in the directory associated with the database
    - File tablename.tdg – maps the trigger to the correspnoding table
    - Triggername.trn contains the trigger definition

Trigger Example

- CREATE TRIGGER trigger_after_sailor_insert

  AFTER INSERT ON SAILORS

  FOR EACH ROW

  BEGIN

  INSERT INTO YoungSailors(sid, name, age, rating)

  SELECT sid, name, age, rating

  FROM New.Sailors N

  WHERE New.age <= 18;

  END;

Trigger has access to **NEW** and **OLD** field values

Trigger Example 2

Triggers that insert rows into the table

```
DELIMITER //

CREATE TRIGGER invoices_after_insert
  AFTER INSERT ON invoices
  FOR EACH ROW
BEGIN
    INSERT INTO invoices_audit VALUES
    (NEW.vendor_id, NEW.invoice_number,
     NEW.invoice_total, 'INSERTED', NOW());
END//

CREATE TRIGGER invoices_after_delete
  AFTER DELETE ON invoices
  FOR EACH ROW
BEGIN
    INSERT INTO invoices_audit VALUES
    (OLD.vendor_id, OLD.invoice_number,
     OLD.invoice_total, 'DELETED', NOW());
END//
```

Reviewing your trigger

- Go to the trigger directory and read the file (.trg)

Program Data\MySQL\MySQL5.7\data\<db-name>*.trg

- Use the DBMS to locate the trigger for you

Triggers in current schema

SHOW TRIGGERS;

ALL Triggers in DBMS using the System Catalog

```
SELECT * FROM Information_Schema.Triggers
WHERE Trigger_schema = 'database_name' AND
  Trigger_name = 'trigger_name';

SELECT trigger_schema, trigger_name, action_statement FROM
information_schema.triggers;
```

Trouble with Triggers

- Action can trigger multiple triggers
  - Execution of the order of the triggers is arbitrary
- Challenge: Trigger action can fire other triggers
  - Very difficult to reason about what exactly will happen
    - Trigger can fire “itself” again
  - Unintended effects possible
- Introducing Triggers leads you to deductive databases
  - Need rule analysis tools that allow you to deduce truths about the data

MySQL limits to triggers

- Triggers not introduced until 5.0
- Not activated for foreign key actions
- No triggers on the MySQL system database
- Active triggers are not notified when the meta data of the table is changed while it is running
- **No recursive triggers**
- Triggers cannot modify/alter the table that is already being used
  - For example the table that triggered it

Changing your trigger

- There is no edit of a trigger
  - CREATE TRIGGER …
  - DROP TRIGGER <TRIGGERNAME>;
  - CREATE TRIGGER …

Events

- MySQL Events are tasks that run according to a schedule.
- An event performs a specific action
- This action consists of an SQL statement, which can be a compound statement in a BEGIN END block
- An event's timing can be either one-time or reoccurring
  - If reoccurring, it can state an interval that determines how often it gets run
  - Can specify a time window to state when the event is active
- An event is uniquely identified by its name and the schema to which it is assigned
- An event is executed with the privileges of its definer/author
- Errors and warnings from an event are written to the log

Events

- CREATE EVENT `event_name`
  ON SCHEDULE *schedule*
  [ON COMPLETION [NOT] PRESERVE]
  [ENABLE | DISABLE | DISABLE ON SLAVE] -- CLUSTERdb
- DO BEGIN
- -- event body
- END

- DROP EVENT `event_name`
- ALTER EVENT `event_name`

Options for a Schedule

- **Run once on a specific date/time:**

  AT ‘YYYY-MM-DD HH:MM.SS’

  e.g. AT ‘2011-06-01 02:00.00’

- **Run once after a specific period has elapsed:**

  AT CURRENT_TIMESTAMP + INTERVAL n [HOUR|MONTH|WEEK|DAY|MINUTE]

  e.g. AT CURRENT_TIMESTAMP + INTERVAL 1 DAY

- **Run at specific intervals forever:**

  EVERY n [HOUR|MONTH|WEEK|DAY|MINUTE]

  e.g. EVERY 1 DAY

- **Run at specific intervals during a specific period:**

  EVERY n [HOUR|MONTH|WEEK|DAY|MINUTE] STARTS date ENDS date

  e.g. EVERY 1 DAY STARTS CURRENT_TIMESTAMP + INTERVAL 1

- WEEK ENDS ‘2017-01-01 00:00.00’

Event example 1

```
DELIMITER $$

CREATE EVENT `archive_blogs`
            ON SCHEDULE EVERY 1 WEEK STARTS '2015-07-24 03:00:00'

DO BEGIN -- copy deleted posts

 INSERT INTO blog_archive (id, title, content)
   SELECT id, title, content FROM blog WHERE deleted = 1;
   -- copy associated audit records
 INSERT INTO audit_archive (id, blog_id, changetype, changetime)
   SELECT audit.id, audit.blog_id, audit.changetype, audit.changetime
     FROM audit JOIN blog ON audit.blog_id = blog.id WHERE blog.deleted = 1;
     -- remove deleted blogs and audit entries
  DELETE FROM blog WHERE deleted = 1;

 END $$


-- reset the delimiter
DELIMITER ;
```

Event example 2

A statement that creates a one-time event

```
DELIMITER //

CREATE EVENT one_time_delete_audit_rows
ON SCHEDULE AT NOW() + INTERVAL 1 MONTH
DO BEGIN
  DELETE FROM invoices_audit
  WHERE action_date < NOW() - INTERVAL 1 MONTH;
END//
```

Event Example 3

A statement that creates a recurring event

```
CREATE EVENT monthly_delete_audit_rows
ON SCHEDULE EVERY 1 MONTH
STARTS '2015-06-01'
DO BEGIN
  DELETE FROM invoices_audit
  WHERE action_date < NOW() - INTERVAL 1 MONTH;
END//
```

Managing events

A statement that disables an event

```
ALTER EVENT monthly_delete_audit_rows DISABLE
```

A statement that enables an event

```
ALTER EVENT monthly_delete_audit_rows ENABLE
```

A statement that renames an event

```
ALTER EVENT one_time_delete_audit_rows
    RENAME TO one_time_delete_audits
```

A statement that drops an event

```
DROP EVENT monthly_delete_audit_rows
```

A statement that drops an event only if it exists

```
DROP EVENT IF EXISTS monthly_delete_audit_rows
```

Prepared statements

- Can create a SQL statement where certain values within the query are parameterized
  - Parameters can be table names, field names, literal values
- Can protect the database against SQL injection
  - Since the structure of the query is defined via the statement
  - Not just free form SQL code
- Less overhead for parsing the statement each time it is executed
- Statement is set up (known to the server)
  - Change the input values to the statement
- The scope of a prepared statement is the session within which it is created

Preparing SQL statement

- Use **PREPARE** to prepare a SQL statement
  - SYNTAX: PREPARE *statementname* from SQLStatement
  - Defines a name from the SQLStatement
  - Within SQLStatement, ? characters denote parameter markers to indicate where data values are to be bound to within the query when it is executed
- Use **EXECUTE** to execute the command
  - SYNTAX: **EXECUTE** SQLStatement [**USING**

    `@var_name [, @var_name] ...`
  - Parameter values can be supplied only by user variables, and the USING clause must name exactly as many variables as the number of parameter markers in the statement
- Use **DEALLOCATE** to free resources associated with the statement

Prepared Statement Example

```
USE scratch; --  using the scratch database
SET @a := "a";
SET @b := "test";
SET @c := 1;
SET @s := CONCAT (“SELECT ", @a, " FROM ", @b,
                                        “WHERE a > " , @c);
PREPARE stmt FROM @s;
EXECUTE stmt; -- can be executed with different values
SET @c := -1;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Summary

- Triggers respond to changes in the database
  - Allows you to define constraints on the data
- Events allow you to schedule tasks to be done by a calendar date or an interval
- Prepared statement allows you to specify the structure of a SQL statement and change literal values passed to the statement.