Database Systems Lab

Lab#11 – Triggers

Triggers

A trigger is procedural SQL code that is automatically invoked by the RDBMS upon the occurrence of a given data manipulation event. It is useful to remember that:

- A trigger is invoked before or after a data row is inserted, updated, or deleted.
- A trigger is associated with a database table.
- Each database table may have one or more triggers.
- A trigger is executed as part of the transaction that triggered it.

In order to explain triggers, let us create an example database for a blogging application. Two tables are required:

1. \`blog\`: stores a unique post ID, the title, content, and a deleted tag.
2. \`audit\`: stores a basic set of historical changes with a record ID, the blog post ID, the change type (NEW, EDIT or DELETE) and the date/time of that change.

The following SQL creates the \`blog\` and indexes the deleted column:

```
CREATE TABLE `blog` (
`id` mediumint(8) unsigned NOT NULL AUTO_INCREMENT,
`title` text,
`content` text,
`deleted` tinyint(1) unsigned NOT NULL DEFAULT '0',
PRIMARY KEY (`id`),
KEY `ix_deleted` (`deleted`)
);
```

The following SQL creates the \`audit\` table. All columns are indexed, and a foreign key is defined for audit.blog_id which references blog.id. Therefore, when we physically DELETE a blog entry, its full audit history is also removed.

```
CREATE TABLE `audit` (
`id` mediumint(8) unsigned NOT NULL AUTO_INCREMENT,
`blog_id` mediumint(8) unsigned NOT NULL,
`changetype` enum('NEW','EDIT','DELETE') NOT NULL,
`changetime`  timestamp  NOT  NULL  DEFAULT  CURRENT_TIMESTAMP  ON  UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
KEY `ix_blog_id` (`blog_id`),
KEY `ix_changetype` (`changetype`),
KEY `ix_changetime` (`changetime`),
CONSTRAINT  `FK_audit_blog_id`  FOREIGN  KEY  (`blog_id`)  REFERENCES  `blog`
(`id`) ON DELETE CASCADE ON UPDATE CASCADE
);
```

When a record is INSERTed into the blog table, we want to add a new entry into the audit table containing the blog ID and a type of 'NEW' (or 'DELETE' if it was deleted immediately).
When a record is UPDATEd in the blog table, we want to add a new entry into the audit table containing the blog ID and a type of 'EDIT' or 'DELETE' if the deleted flag is set.
Note that the changetime field will automatically be set to the current time.
Each trigger requires:

1. A unique name. It is preferred to use a name which describes the table and action, e.g. blog_before_insert or blog_after_update.
2. The table which triggers the event. A single trigger can only monitor a single table.
3. When the trigger occurs. This can either be BEFORE or AFTER an INSERT, UPDATE or DELETE. A BEFORE trigger must be used if you need to modify incoming data. An AFTER trigger must be used if you want to reference the new/changed record as a foreign key for a record in another table.
4. The trigger body; a set of SQL commands to run. Note that you can refer to columns in the subject table using OLD.col_name (the previous value) or NEW.col_name (the new value). The value for NEW.col_name can be changed in BEFORE INSERT and UPDATE triggers.

The basic trigger syntax is:

```
CREATE TRIGGER `event_name` BEFORE/AFTER INSERT/UPDATE/DELETE
```

```
ON `database`.`table`
FOR EACH ROW BEGIN
      -- trigger body
      -- this code is applied to every
      -- inserted/updated/deleted row
END;
```

We require two triggers — AFTER INSERT and AFTER UPDATE on the blog table. It’s not necessary to define a DELETE trigger since a post is marked as deleted by setting its deleted field to true.
Our trigger body requires a number of SQL commands separated by a semi-colon (;). To create the full trigger code, we must change delimiter to something else such as $$.
Our AFTER-INSERT trigger can now be defined. It determines whether the deleted flag is set, sets the @changetype variable accordingly, and inserts a new record into the audit table:

```
DELIMITER $$
CREATE
      TRIGGER `blog_after_insert` AFTER INSERT
      ON fb.`blog`
      FOR EACH ROW BEGIN
            IF NEW.deleted THEN
                  SET @changetype = 'DELETE';
            ELSE
                  SET @changetype = 'NEW';
            END IF;

            INSERT  INTO  audit  (blog_id,  changetype)  VALUES  (NEW.id,
@changetype);
END  $$
DELIMITER ;
```

The AFTER UPDATE trigger is almost identical:

```
DELIMITER $$

CREATE
```

```
      TRIGGER `blog_after_update` AFTER UPDATE
      ON fb.`blog`
      FOR EACH ROW BEGIN
            IF NEW.deleted THEN
                  SET @changetype = 'DELETE';
            ELSE
                  SET @changetype = 'EDIT';
            END IF;

            INSERT  INTO  audit  (blog_id,  changetype)  VALUES  (NEW.id,
@changetype);
END $$

DELIMITER ;
```

Let us see what happens when we insert a new post into our blog table:

```
INSERT INTO blog (title, content) VALUES ('Article One', 'Initial text.');
```

Check both the blog and the audit table.

Now let us update our blog text:

```
UPDATE blog SET content = 'Edited text' WHERE id = 1;
```

Check the blog and audit tables again.

Finally, let us mark the post as deleted:

```
UPDATE blog SET deleted = 1 WHERE id = 1;
```

The `audit` table is updated accordingly and we have a record of when changes occurred.

For more details:
https://www.w3resource.com/mysql/mysql-triggers.php