INFO2820 – Database Systems I (Adv)

Week 7: Trigger Execution Details

Dr. Uwe Röhm
School of Information Technologies

Outline

- Design Space of Triggers
- Short Detour: Stored Procedures in PostgreSQL
- Understanding Trigger Executions
- Trigger Specialities from Oracle and MySQL

Design Space of Triggers

- **Activation** - Occurrence of the *event*
- **Consideration** - The point, after activation, when *condition* is evaluated
  - Immediate or deferred (when the transaction requests to commit)
  - *Condition* might refer to both the state before and the state after *event* occurs
- **Execution** – point at which *action* occurs
  - With deferred consideration, execution is also deferred
  - With immediate consideration, execution can occur immediately after consideration or it can be deferred
    - If execution is immediate, execution can occur before, after, or instead of triggering event.
    - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.
- **Granularity**
  - *per-row granularity* or *per-statement granularity*

Triggers in SQL:1999

- **Events**: INSERT, DELETE, or UPDATE statements or changes to individual rows caused by these statements
  - Some DBMS also allow DDL triggers or TRUNCATE
- **Condition**: Anything that is allowed in a WHERE clause
- **Action**: An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM) (which can contain embedded SQL statements)
- **Consideration**: *Immediate*
  - Condition can refer to both the state of the affected row or table before *and* after the event occurs
  - Some systems (e.g. postgres) allow deferred consideration too
- **Execution**: *Immediate* – but can be **before** or **after** the execution of the triggering event
  - Action of before trigger cannot modify the database
  - Since SQL:2008: also INSTEAD OF triggers
    - Any idea when this could become handy?
- **Granularity**: Both *row-level* and *statement-level*

Triggers – PostgreSQL Syntax

```
CREATE TRIGGER trigger-name

  BEFORE      INSERT
  AFTER       DELETE                 ON relation-name
  INSTEAD OF  UPDATE
              TRUNCATE
  FOR EACH ROW                 -- optional; only for row-triggers
                                -- optional; otherwise a statement trigger
  WHEN ( condition )           -- optional
  EXECUTE PROCEDURE stored-procedure-name;-- needs to be defined 1st
```

-- PL/pgSQL can be used to define trigger procedures
-- needs to be specified with no arguments
-- When a PL/pgSQL function is called as a trigger, several special variables
-- are created automatically in the top-level block:

NEW
OLD
TG_WHEN ('BEFORE' or 'AFTER')
TG_OP ('INSERT', 'DELETE, 'UPDATE', 'TRUNCATE')
…

[cf. http://www.postgresql.org/docs/9.2/static/plpgsql-trigger.html
http://www.postgresql.org/docs/9.2/static/sql-createtrigger.html]

BeforeTrigger Example
(row-level granularity, PostgreSQL syntax)

```
CREATE TABLE Foo( …, lock BOOLEAN NOT NULL DEFAULT FALSE );

CREATE FUNCTION LockTrigger() RETURNS trigger AS $$
BEGIN
   IF OLD.lock = TRUE AND !isUpdate(Foo.lock) THEN
      RAISE EXCEPTION 'tuple is locked'; -- aborts
   END IF
   RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER Foo_TupleLock
    BEFORE UPDATE OR DELETE ON Foo
    FOR EACH ROW
    EXECUTE PROCEDURE LockTrigger();
```

Make sure the affected rows are not 'locked'

After Trigger Example

(statement granularity, PostgreSQL syntax)

```
CREATE TABLE Log ( … );
CREATE FUNCTION SalaryLogger() RETURNS trigger AS $$
BEGIN
   INSERT INTO Log
         VALUES (CURRENT_DATE, SELECT AVG(Salary)
                                     FROM Employee );
   RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER RecordNewAverage
    AFTER UPDATE OF Salary ON Employee
    FOR EACH STATEMENT
    EXECUTE PROCEDURE SalaryLogger();

SELECT * FROM INFORMATION_SCHEMA.TRIGGERS;
```

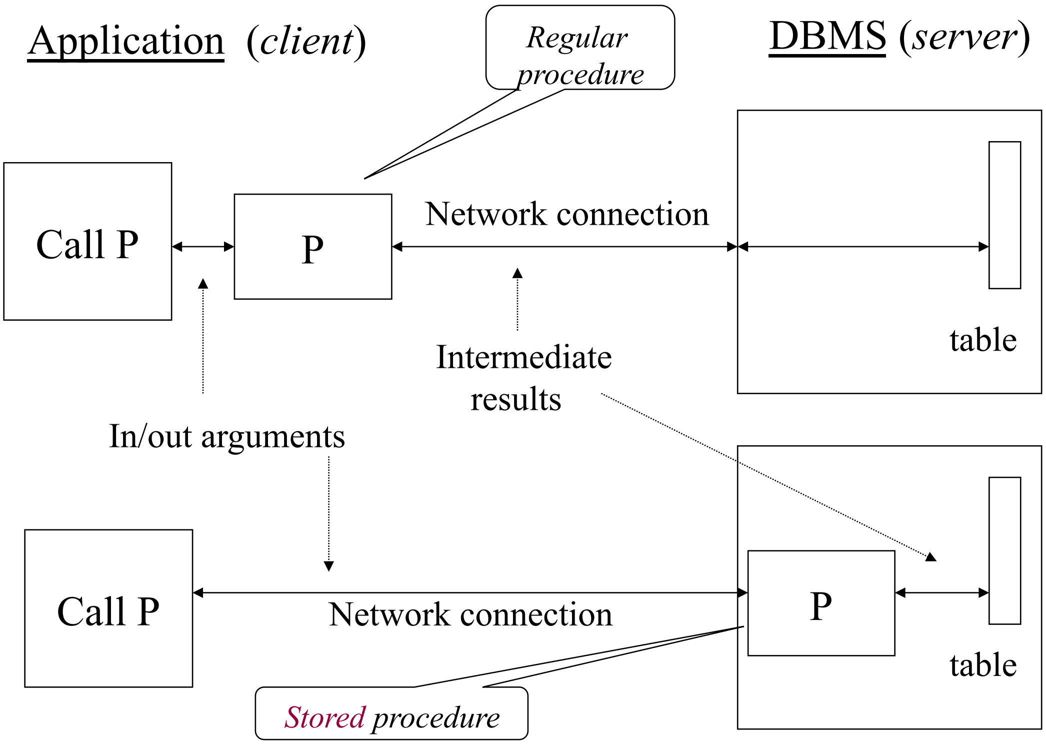
Keep track of salary averages in the log

Check which triggers exist.

Detour: Stored Procedures

- Any major DBMS allows user-defined functions to be defined and executed inside the DBMS
  - So-called *stored procedures*
- Benefits:
  - Execution on the database server close to data
    - no need for data transfer of network or multiple requests
  - Central code base as part of DBMS
  - Extendible query execution
    - Some systems allow functions and aggregate functions to be defined and then be used as part of SELECT queries
- In PostgreSQL:
  - A stored 'procedure' is actually always a 'stored function'
    - Can take arguments and can return a return value (even a table!)
  - Specialty: multiple languages supported!
    - As of 9.1: SQL, PL/pgSQL, PL/Tcl, PL/Perl, PL/Python, external C

Detour: Stored Procedures



PL/pgSQL Example

(cf. http://www.postgresql.org/docs/9.2/static/plpgsql-structure.html)

Tip: CREATE OR REPLACE to avoid 'name-already-used'

- PL/pgSQL procedure declaration

```
CREATE OR REPLACE FUNCTION
   name ( parameter1, …, parameterN ) RETURNS sqlType
AS $$
DECLARE
  variable   sqlType;
  …
BEGIN
    …
END;
$$ LANGUAGE plpgsql;
```

optional

Tip: final delimiter must match the one used after AS

Different languages supported;

Detour: PostgreSQL Function Examples

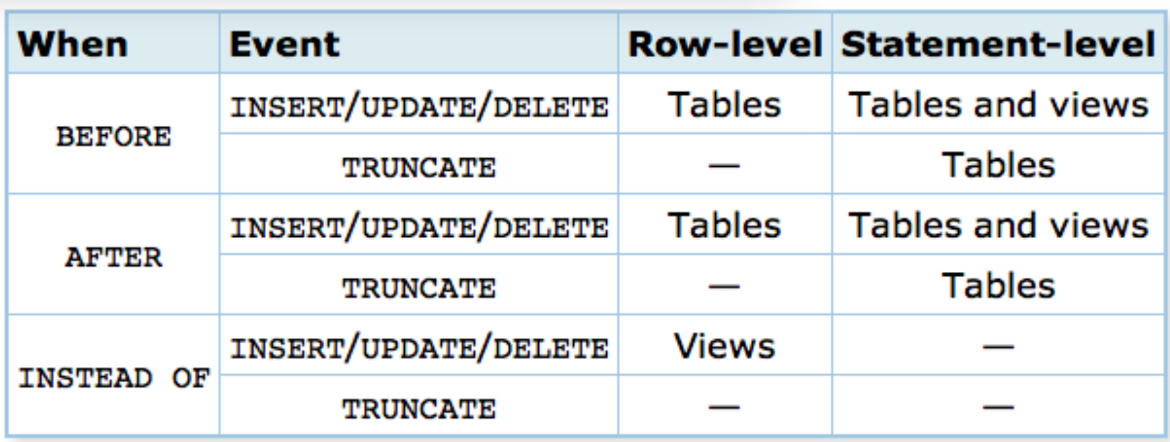
```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
integer AS $incr$
     BEGIN
           RETURN i + 1;
     END;
$incr$ LANGUAGE plpgsql;


CREATE FUNCTION add(integer, integer) RETURNS integer
   AS 'SELECT $1 + $2;'
LANGUAGE SQL
IMMUTABLE RETURNS NULL ON NULL INPUT;

// cf. http://www.postgresql.org/docs/9.2/static/trigger-example.html
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'  LANGUAGE C;
```

Triggers and Views

- INSTEAD OF triggers are interesting to make even complex views updatable.
- The following table summarizes when and on which data object each type of trigger can be used in PostgreSQL 9:



When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

http://www.postgresql.org/docs/9.2/static/sql-createtrigger.html

INSTEAD OF Trigger Example

The scenario:

```
CREATE TABLE Person (
   name VARCHAR(20) PRIMARY KEY,
   address VARCHAR(50)
);
CREATE TABLE StudentTable (
   sid INTEGER PRIMARY KEY,
   name VARCHAR(20) REFERENCES Person,
  degree VARCHAR(10)
);
CREATE VIEW Students AS
          SELECT sid, P.name AS name, P.address, degree
             FROM Person P, StudentTable S
           WHERE P.name = S.name;

-- Try:  INSERT INTO Students VALUES (001, 'Peter', 'Neverland', 'B.I.T.');
```

INSTEAD OF Trigger Example (cont'd)

```
CREATE FUNCTION StudentViewInsert() RETURNS trigger AS
$body$
BEGIN
          IF (NOT EXISTS (SELECT 1 FROM Person WHERE P.name = NEW.name)) THEN
                    INSERT INTO Person VALUES (NEW.name, NEW.address);
          END IF;
          IF (NOT EXISTS (SELECT 1 FROM StudentTable WHERE S.sid=NEW.sid) ) THEN
                    INSERT INTO StudentTable VALUES (NEW.sid, NEW.name, NEW.degree);
          ELSE
                    -- If duplicate, change to UPDATE so that there will not
                    -- be a duplicate key violation error.
                    UPDATE StudentTable SET degree=NEW.degree WHERE sid=NEW.sid;
          END IF;
          RETURN NEW;
END
$body$ LANGUAGE plpgsql;

CREATE TRIGGER INS_Trigger_Students
          INSTEAD OF INSERT ON Students
          FOR EACH ROW EXECUTE PROCEDURE StudentViewInsert();
```

Some Tips on Triggers

- Use BEFORE triggers
  - For checking complex integrity constraints and modifying rows
- Use AFTER triggers
  - For integrity maintenance and update propagation
- In Oracle, triggers cannot access “mutating” tables
  - e.g. AFTER trigger on the same table which just updates

- Good overviews:
  - Kifer/Bernstein/Lewis: “Database Systems - An Application-oriented Approach”, 2nd edition, Chapter 7.
  - Michael v.Mannino: “Database - Design, Application Development and Administration”
  - PostgreSQL Manual, Chapter 36 – Triggers & Chapter 39 – PL/pgSQL
  - Oracle Application Developer’s Guide, Chapter 15

Understanding Trigger Execution

- Individual triggers are easy to understand.
- **Multiple Triggers** however….
  - How should multiple triggers activated by a single event be handled?
    - Evaluate one condition at a time and if true immediately execute action or
    - Evaluate all conditions, then execute actions
  - The execution of an action can affect the truth of a subsequently evaluated condition so the choice is significant.
- What effect do integrity constraint enforcement and database actions have on trigger execution?

Example: What Happens Here?

```
CREATE TABLE WhenOffered (
  uosCode       CHAR(8),
  semester      CHAR(2) CHECK (semester IN ('S1','S2')),
  PRIMARY KEY (uosCode, semester)
);

CREATE FUNCTION SemesterUpdater() RETURNS trigger AS $$
BEGIN
   NEW.semester := 'S3';
   RETURN NEW;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER OfferFix
   AFTER INSERT ON WhenOffered
   FOR EACH ROW EXECUTE PROCEDURE SemesterUpdater();

INSERT INTO WhenOffered VALUES ('INFO2120', 'S2');
```

Trigger Execution Procedure

- Simplified Oracle & Postgres Trigger Execution Procedure:
  1. Execute the applicable BEFORE STATEMENT triggers
  2. For each row affected by the SQL statement:
     a) Execute the applicable BEFORE ROW triggers
        a) In Postgres, can return NEW row or NULL -> stops execution of statement
     b) Perform data manipulation operation on the row
     c) Perform integrity constraint checking
     d) Execute applicable AFTER ROW triggers
  3. Perform deferred integrity constraint checking
  4. Execute applicable AFTER STATEMENT triggers
- Notes:
  - A trigger is *applicable* if it contains an event that matches the statement type and its WHEN condition is TRUE
  - Most constraint checking occurs after BEFORE ROW triggers
  - Deferred constraint checking is performed at end of transaction
    - In most applications, only few constraints are declared deferred

WHEN Condition

- WHEN condition is part of the consideration whether a trigger is executed at all
- BEFORE triggers:
  - checked before the actual trigger execution and also before the corresponding insert/delete/update statement
  - Hence no big difference whether check is expressed in WHEN or at the start of the actual trigger function
- AFTER trigger:
  - Checked after the corresponding row has been updated / the whole statement has been executed, but again before the trigger runs
  - Can now make a **big performance improvement** to check condition as part of WHEN rather than in the trigger if the selectivity of the trigger is high (only few rows actually fire the trigger)
    - DBMS does not need to initialize and execute any trigger function if WHEN condition is not TRUE

Overlapping Triggers

- two or more triggers with the same timing, granularity, and applicable table, that an SQL statement may cause both triggers to fire
- Example:
  - BEFORE ROW trigger on UPDATE ON Customer
    overlaps with
    BEFORE ROW trigger on UPDATE OF CustBal ON Customer
- Problem: What is the execution order of triggers?
  - SQL:1999 standard:
    execution order depends on time when triggers are defined;
    triggers are executed in the order in which they were created
  - Oracle: execution order is arbitrary!
  - PostgreSQL: execution order is alphabetical by trigger name!
    - At least documented as such for 'triggers on the *same* event'

After Trigger Example
(row granularity, Oracle Syntax)

No salary raises greater than 5%

```
CREATE TRIGGER LimitSalaryRaise
   AFTER UPDATE OF Salary ON Employee
   REFERENCING OLD AS O
               NEW AS N
   FOR EACH ROW
   WHEN (N.Salary-O.Salary > 0.05*O.Salary)
   BEGIN
        UPDATE Employee                -- action
        SET Salary = 1.05 * O.Salary
        WHERE Id = O.Id;
   END;
```

Note: The action itself is a triggering event (but in this case a chain reaction is not possible)

Recursive Trigger Execution

- Triggers can execute recursively
  - This is also called '*cascading triggers*'
- For example if the action of the trigger contains data manipulation statements itself
- Similar with referential integrity constraints
  - ON *event* CASCADE or SET NULL or SET DEFAULT
  - This causes deletion or updates on related rows in a dependent table
  - Actions on referenced rows can cause triggers to fire leading to recursive execution of triggers
- Most DBMS limit the recursion depth
  - Not so PostgreSQL?
    "There is no direct limitation on the number of cascade levels. … It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios."

Trigger Execution Procedure

- Full Oracle Trigger Execution Procedure:
  1. Execute the applicable BEFORE STATEMENT triggers
  2. For each row affected by the SQL statement:
     a) Execute the applicable BEFORE ROW triggers.
        Recursively execute the procedure for each data manipulation in trigger
     b) Perform data manipulation operation on the row
     c) Perform integrity constraint checking;
        Recursively execute the procedure for actions on referenced rows.
     d) Execute applicable AFTER ROW triggers;
        Recursively execute the procedure for each data manipulation in trigger
  3. Perform deferred integrity constraint checking
  4. Execute applicable AFTER STATEMENT triggers

Some Guidelines on Triggers

- Do not use data manipulation statements (such as insert, delete or update) in BEFORE triggers
  - Try to restrict manipulation to just the current row
- Limit data manipulation statements in AFTER triggers
- For triggers that fire on UPDATE statements, always list the columns on which the trigger applies
- Ensure that overlapping triggers do not depend on a specific order to fire.
  - In most DBMS the firing order is arbitrary
- Be cautious about triggers on tables affected by actions on referenced rows. These triggers will fire as a result of actions on the parent tables.

Some Specialities with Oracle Triggers

- Events can also be DDL statements
  - Example: audit any CREATE statements in your schema

```
CREATE TRIGGER audit_db_object
         AFTER CREATE  ON SCHEMA
         pl/sql_block
```

  - Example: React to login errors

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
BEGIN
  IF (IS_SERVERERROR (1017)) THEN
      <special processing of logon error>
   ELSE
      <log error number>
   END IF;
END;
```

- Restriction on WHEN condition:
  - Oracle does not support SQL queries in WHEN condition of triggers
- INSTEAD OF triggers can be used to make views updatable
  - If a view is already updatable, the INSTEAD OF trigger precedes

Oracle: Autonomous Transaction

- Normally, stored procedures and triggers are executed as part of the same global transaction
  - Any error or integrity constraint violation within their code aborts the overall transaction
- Oracle provides a facility to run triggers or stored procedurers as separate, autonomous transactions
- Example:

```
CREATE TRIGGER anniversary_trigger
   BEFORE INSERT ON employees FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO anniversaries VALUES(TRUNC(:new.hire_date));
   -- Only commits the preceding INSERT, not the INSERT that fired
   COMMIT;
   EXCEPTION
-- If someone else was hired on the same day, we get an exception
-- because of duplicate values. That's OK, no action needed.
      WHEN OTHERS THEN NULL;
END;
```

Triggers in MySQL

- MySQL supports triggers since version 5.0.2
- Commands
  - CREATE TRIGGER
  - DROP TRIGGER
  - SHOW TRIGGERS
- Several restrictions:
  - as of v5.6, MySQL does not support statement-level triggers
  - events: INSERT, DELETE or UPDATE
    - update trigger can't specify single attributes, scope always whole table
  - no WHEN condition clause
  - action can be specified in the MySQL stored procedure language
  - Cf. http://dev.mysql.com/doc/refman/5.6/en/create-trigger.html

Triggers – MySQL Example

```
mysql> delimiter //
mysql> 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;//
mysql> delimiter ;

mysql> SELECT * FROM INFORMATION_SCHEMA.TRIGGERS;
mysql> SHOW TRIGGERS LIKE '%_check'
```