TRIGGERS

Dirk Van Gucht

TRIGGERS

- A trigger is a user-defined function that is triggered (invoked) when state-changes are applied to a relation or a view with which the trigger is associated
- State changes can be INSERT, DELETE, or UPDATE statements
- Triggers can be applied in different temporal modes:

  BEFORE
  AFTER
  INSTEAD
- Triggers can be applied by row level or by single statement level
- We will be mainly concerned with row level triggers

APPLICATIONS OF TRIGGERS

- Materialized view
- View updates
- Constraint verification
- Dynamic aggregate function values maintenance
- Logging/auditing

TRIGGER STRUCTURE

- The programming structure consists of two components:
- (1) A trigger function
- (2) A trigger definition

TRIGGER FUNCTION

- A trigger function is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.
- When a function is called as a trigger, several special variables are created automatically. Two important variables are:
  - NEW: variable holding the new database row for INSERT/UPDATE operations in row-level triggers.
  - OLD: variable holding the old database row for UPDATE/DELETE operations in row-level triggers.

EXAMPLE: *INSERT INTO TRIGGER* ON A RELATION

- We will consider an INSERT trigger on the relation Student(sid, sname, major, byear)
- We assume there is a relation CS_student(sid,sname,byear) which holds the information about 'CS' majors
- Objective: an INSERT INTO Student should trigger an INSERT INTO the CS_student relation if the student is a 'CS' major

EXAMPLE: INSERT INTO TRIGGER ON A RELATION

- Trigger function (describes trigger action to be done)

```
CREATE OR REPLACE FUNCTION insert_into_Student() RETURNS TRIGGER AS
 $$
  BEGIN
   IF NEW.Major = 'CS' THEN
     INSERT INTO CS_Student VALUES (NEW.sid, NEW.sname, NEW.byear);
   END IF;
   RETURN NEW;
  END;
 $$ LANGUAGE 'plpgsql';
```

- The variable NEW is a system-provided variable that refers to the tuple that is inserted into Student relation

EXAMPLE: INSERT INTO TRIGGER ON A RELATION

Trigger definition (declaration) on the Student relation

```
CREATE TRIGGER insert_into_Student_Relation
   BEFORE INSERT ON Student
   FOR EACH ROW
      EXECUTE PROCEDURE insert_into_Student();
```

BEFORE INSERT tells us that an action needs to be executed before the tuple is inserted into Student

That action is specified by the code in the trigger function insert_into_Student()

EFFECTS

INSERT INTO Student VALUES ('s5', 'Linda', 'Math', 1960)

will insert this tuple only in the Student relation

INSERT INTO Student VALUES ('s6', 'Marc', 'CS', 1963)

will insert the tuple ('s6', 'Marc', 1963) in the CS_course relation first (BEFORE) and then it

will insert this tuple ('s6', 'Marc', 'CS', 1963) in the Student relation

FOR EACH ROW CLAUSE

- The FOR EACH CLAUSE guarantees that the trigger function is applied for each possible tuple that can be assigned to the NEW variable

FOR EACH ROW SEMANTICS

- Multiple inserts will be handled one at a time

```
INSERT INTO Student VALUES ('s7', 'Nick', 'CS', 1990),
                           ('s8', 'Vince', 'Biology', 1985);
```

- The NEW variable will be assigned to the first of these tuples
- The trigger function will work on that NEW variable
- Subsequently, the NEW variable will be assigned to the second of these tuples
- The trigger function will then work on this NEW variable

- Both tuples will be inserted into the Student relation
- But only the tuple ('s7', 'Nick', 1990) will be inserted into the CS_student relation

RETURNING NEW VERSUS NULL

- Consider a small change to the trigger function
- RETURN NULL instead of RETURN NEW

```
CREATE OR REPLACE FUNCTION insert_into_Student() RETURNS
TRIGGER AS
  $$
    BEGIN
    IF NEW.Major = 'CS' THEN
      INSERT INTO CS_Student VALUES (NEW.sid, NEW.sname,
NEW.byear);
    END IF;
    RETURN NULL;
    END;
  $$ LANGUAGE 'plpgsql';
```

- The insertion of a tuple into Student will be ignored
- The insertion of a tuple in CS_student will still be done

BEFORE VERSUS AFTER

- With AFTER trigger on an insert into Student, the insert will always succeed before the trigger function executes
- In the AFTER case, a RETURN NEW statement will do nothing.
- So it is typical to return NULL for AFTER triggers

AFTER TRIGGER

```
CREATE OR REPLACE FUNCTION insert_into_Student() RETURNS TRIGGER
AS
 $$
   BEGIN
   IF NEW.Major = 'CS' THEN
     INSERT INTO CS_Student VALUES (NEW.sid, NEW.sname NEW.byear);
   END IF;
   RETURN NULL; -- Equivalent with RETURN NEW
   END;
 $$ LANGUAGE 'plpgsql';

CREATE TRIGGER insert_into_Student_Relation
   AFTER INSERT ON Student
   FOR EACH ROW
   EXECUTE PROCEDURE insert_into_Student();
```

DELETE TRIGGER (OLD VARIABLE)

```
CREATE OR REPLACE FUNCTION delete_from_Student()
  RETURNS TRIGGER AS
 $$
   BEGIN
     IF OLD.Major = 'CS' THEN
         DELETE FROM CS_student WHERE sid = OLD.sid;
     END IF;
     RETURN OLD;
   END;
 $$ LANGUAGE 'plpgsql';

CREATE TRIGGER delete_from_Student_Relation
   BEFORE DELETE ON Student
   FOR EACH ROW
   EXECUTE PROCEDURE delete_from_Student();
```

FOR EACH ROW CLAUSE

- The FOR EACH CLAUSE guarantees that the trigger function is applied for each possible tuple that can be assigned to the OLD variable

VIEW UPDATE (INSERT)

- Base relation Student(Sid, Sname, Major,Byear)
- Consider a view for CS_student

```
CREATE VIEW CS_Student AS
    SELECT  sid, sname, byear
    FROM    Student
    WHERE Major = 'CS'
```

- Consider the insertion of new CS students into the CS_student view
- The desired effect is to insert these new students into the base relation Student and specify their major as 'CS'

VIEW UPDATE (INSERT)

INSERT INTO CS_Student VALUES ('s1', 'Eric', 1990);

should "trigger"

INSERT INTO Student VALUES('s1', 'Eric', 'CS', 1990);

Remark: Notice that we can always insert into the Student relation

TRIGGER FUNCTION DEFINITION

- We first define a trigger function that accomplishes each insert into the base relation Student

```
CREATE FUNCTION insert_CS_student() RETURNS TRIGGER AS
  $$
  BEGIN
    INSERT INTO Student VALUES(NEW.sid, NEW.sname, `CS' , NEW.byear);
    RETURN NULL;
  END;
  $$ LANGUAGE `plpgsql';
```

TRIGGER DEFINITION FOR INSERT INTO VIEW

- After the trigger function has been defined, we bind it to a trigger declared for the CS_Student view

```
CREATE TRIGGER insert_into_CS_student
    INSTEAD OF INSERT ON CS_Student
    FOR EACH ROW
    EXECUTE PROCEDURE insert_CS_Student();
```

- Notice that we must use INSTEAD

VIEW UPDATE (DELETE)

- Consider the deletion of students from the CS_student view
- The desired effect is to delete those students from the base relation Student

```
CREATE OR REPLACE FUNCTION delete_CS_student() RETURNS TRIGGER AS
  $$
  BEGIN
    DELETE FROM Student WHERE sid = OLD.sid;
    RETURN NULL;
  END;
  $$ LANGUAGE 'plpgsql';
```

TRIGGER FUNCTION AND TRIGGER DEFINITIONS FOR DELETION FROM VIEW

- Trigger function

```
CREATE FUNCTION delete_CS_student() RETURNS TRIGGER AS
  $$
  BEGIN
    DELETE FROM Student WHERE sid = OLD.sid;
    RETURN NULL;
  END;
  $$ LANGUAGE 'plpgsql';
```

- Trigger

```
CREATE TRIGGER delete_from_CS_student
   INSTEAD OF DELETE ON CS_Student
   FOR EACH ROW
   EXECUTE PROCEDURE delete_CS_Student();
```

VIEW UPDATE (DELETE)

- OLD is a system-maintained tuple variable that refers to any tuple that is to be deleted from the CS_Student view

```
DELETE FROM CS_Student WHERE sname = 'John';
```

This should trigger deleting all students with name 'John' and major 'CS' from the Student table

```
DELETE FROM Student WHERE sname = 'John' AND
                          major  = 'CS';
```

VIEW MATERIALIZATION

- Consider maintaining a relation Math_Student that is the materialization of a view for Math students

```
CREATE TABLE Math_Student
  (Sid INTEGER,
   Sname TEXT);
```

MATERIALIZATION: TRIGGER FUNCTION AND TRIGGER DEFINITIONS

```
CREATE FUNCTION insert_Math_student() RETURNS TRIGGER AS
  $$
  BEGIN
    IF (NEW.Major = 'Math')
      THEN INSERT INTO Math_Student VALUES(NEW.sid, NEW.sname);
    END IF;
    RETURN NULL;
  END;
  $$ LANGUAGE 'plpgsql';


CREATE TRIGGER add_Math_Student
   AFTER INSERT ON Student
   FOR EACH ROW
   EXECUTE PROCEDURE insert_Math_Student();
```

CONSTRAINT VERIFICATION (KEYS)

- Ensure that sid is a primary key

```
CREATE OR REPLACE FUNCTION check_Student_key_constraint() RETURNS trigger AS
$$
BEGIN
   IF NEW.sid IN (SELECT sid FROM Student) THEN
     RAISE EXCEPTION ‘sid already exists';
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

     CREATE TRIGGER check_Student_key_
          BEFORE INSERT
          ON Student
          FOR EACH ROW
          EXECUTE PROCEDURE check_Student_key_constraint();
```

AGGREGATE FUNCTION MAINTENANCE

- Keep track of number of students

```
CREATE TABLE Count_Students( total integer);
INSERT INTO    Count_Students VALUES(0);

CREATE OR REPLACE FUNCTION Maintain_Number_Students() RETURNS trigger AS
$$
BEGIN
  UPDATE Count_Students SET total = total + 1;
  RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER Total_Students
   AFTER INSERT ON Student
   FOR EACH ROW
   EXECUTE PROCEDURE Maintain_Number_Students();
```

ALTERNATIVE SOLUTION

```
CREATE OR REPLACE FUNCTION Maintain_Number_Students() RETURNS trigger AS
$$
BEGIN
  UPDATE Count_Students SET total = (SELECT COUNT(*) FROM Student);
  RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER Total_Students
   AFTER INSERT ON Student
   EXECUTE PROCEDURE Maintain_Number_Students();
```

- In this case, Total_Students is a STATEMENT trigger:

  the trigger function is executed once independent of

  the number of tuples inserted in the Student relation

MAINTAINING A LOG

- Any time we insert a new student, we want to log a record with the time stamp for that insert

- We will maintain a Student_log relation for this

```
CREATE TABLE Student_log(
    sid text,
    stamp timestamp
);
```

MAINTAINING A LOG

- We can do this with the following trigger

```
CREATE OR REPLACE FUNCTION time_stamp_for_student() RETURNS
TRIGGER AS
 $$
 BEGIN
   INSERT INTO Student_log VALUES (NEW.sid, now() );
   RETURN NULL;
 END;
$$  LANGUAGE plpgsql;

CREATE TRIGGER log_student
  AFTER INSERT ON Student
  FOR EACH ROW EXECUTE PROCEDURE time_stamp_for_student();
```

TRIGGERS CAN LEAD TO INFINITE LOOPS

```
CREATE TABLE A (x INTEGER);

CREATE FUNCTION insert_in_A () RETURNS TRIGGER AS
$$
BEGIN
  INSERT INTO A VALUES(NEW.x);
  RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER insert_into_A_relation
  BEFORE INSERT ON A
  FOR EACH ROW EXECUTE PROCEDURE insert_in_A();

INSERT INTO A VALUES(1);
```

This will lead to an infinite loop;

OTHER ISSUES

- Triggers can be hard to debug
- When multiple triggers are defined on the same relation, the order in which these triggers get evaluated becomes relevant
- Trigger can incur performance overhead
- Triggers are powerful aids in managing and protecting the state of the database in sophisticated ways
- Triggers are a fundamental component of event-driven programming