CSCI235 Database Systems

Database Triggers

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Database Triggers

Outline

Database trigger ? What is it ?

Database trigger is a piece of code stored in a data dictionary and automatically processed whenever a pre-defined event happens and pre-defined condition is satisfied

For example, we would like to automatically increase job level for all employees whose salary is above 100000

Modification of derived data

```
ON UPDATE OF EMPLOYEE.salary
  IF NEW.salary > 100000 THEN
    EXECUTE IncreaseJobLevel(NEW.enumber, NEW.salary);
  END IF;
```

For example, we would like to implement a data security rule saying that a salary cannot be updated over a weekend

Checking a constraint

```
ON UPDATE OF EMPLOYEE.salary
  IF EXTRACT(CURRENT_DATE,'Day')) IN ('Saturday', 'Sunday') THEN
    RAISE EXCEPTION 'Salary cannot be updated over a weekend !';
  END IF;
```

Database trigger ? What is it ?

For example, we would like to enforce a consistency constraint saying that a department cannot have more than 100 employees

```
ON INSERT INTO EMPLOYEE                                          Testing an event
```

```
  SELECT COUNT(*)                                    Checking a consistency constraint
  INTO total_employees
  FROM EMPLOYEE
  WHERE dname = NEW.dname;
```

```
IF total_employees = 100 THEN                                    Performing an action
    RAISE EXCEPTION 'Too many employees in %', NEW.dname);
  END IF;
```

In the example above we assume that a trigger fires and it is processed before `INSERT` statement

Sometimes it is more convenient to fire a trigger that verifies a consistency constraint after modification of a relational table and before `COMMIT` statement

This is why we have two temporal options for triggers: `BEFORE` and `AFTER`

Database trigger ? What is it ?

What do we need database triggers for ?

- To verify the consistency constraints
- To enforce the sophisticated database access controls
- To implement transparent event logging
- To generate the values of derived attributes
- To maintain replicated data in a distributed database
- To update the relational views

Active Database Systems provide functionalities for implementation of database triggers

Database Triggers

Outline

Active database system

Active database system is a database system which is able to detect the events that have happened in a certain period of time and in the response to these events it is able to execute the actions when the pre-defined conditions are met

A logic of active database system is implemented as a collection of Event-Condition-Action (ECA)rules

In SQL ECA rule can be created with `CREATE TRIGGER` statement and it can be deleted with `DROP TRIGGER` statement

Syntax of ECA rule:

- (`EVENT`, `CONDITION`, `ACTION`)

Semantics of ECA rule:

- Whenever an `EVENT` happens and a `CONDITION` is satisfied then a database system performs an `ACTION`

Active database system

A sample event

```
ON UPDATE OF EMPLOYEE.salary
```
Event

A sample condition

```
IF NEW.salary > 100000
```
Condition

A sample Action

```
EXECUTE IncreaseJobLevel(NEW.enumber, NEW.salary);
```
Trigger

`CREATE OR REPLACE TRIGGER` statement implements ECA rule

Database Triggers

Outline

CREATE OR REPLACE TRIGGER statement

A sample `CREATE OR REPLACE TRIGGER` statement

```
CREATE OR REPLACE TRIGGER CheckBudget                    Trigger name
```

Temporal option

```
                                                         Temporal option specification
BEFORE
```

Event

```
                                                         Event specification
UPDATE OF budget ON DEPARTMENT
```

Type of trigger, either statement or row trigger

```
                                                         Trigger type: row
FOR EACH ROW                    -- FOR EACH ROW means that it is a row trigger
```

Action

```
EXECUTE PROCEDURE CheckBudget();                         Trigger condition
```

CREATE OR REPLACE TRIGGER statement

Implementation of action

Start of trigger action

```
CREATE OR REPLACE FUNCTION CheckBudget()
RETURNS TRIGGER AS
$$
BEGIN
```

Variables OLD and NEW represent a row before modification or deletion and a row after modification or insertion

Application of NEW variable in a row trigger

```
  IF NOT ( NEW.budget BETWEEN 1 AND 7000 ) THEN
```

Abnormal termination of a trigger together with a transaction that fired a trigger

Abnormal termination of a trigger

```
    RAISE EXCEPTION 'Budget of department % cannot be equal to %',
                    NEW.name, NEW.budget;
  END IF;
```

CREATE OR REPLACE TRIGGER statement

Normal termination of a trigger together with a transaction that fired a trigger

```
    RETURN NEW;                                    Normal termination of a trigger
```

End of trigger body

```
END;                                               End of trigger action
$$
LANGUAGE plpgsql;
```

CREATE OR REPLACE TRIGGER statement

A complete CREATE OR REPLACE TRIGGER statement

A sample row trigger

```
CREATE OR REPLACE TRIGGER CheckBudget
  BEFORE UPDATE OF budget ON DEPARTMENT
  FOR EACH ROW
    EXECUTE PROCEDURE CheckBudget();
```

Implementation of an action

Action

```
CREATE OR REPLACE FUNCTION CheckBudget()
RETURNS TRIGGER AS
$$
BEGIN
  IF NOT ( NEW.budget BETWEEN 1 AND 7000 ) THEN
    RAISE EXCEPTION 'Budget of department % cannot be equal to %',
                    NEW.name, NEW.budget;
  END IF;
  RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

CREATE OR REPLACE TRIGGER statement

Another implementation of the same trigger with `WHEN` clause

A sample row trigger

```
CREATE OR REPLACE TRIGGER CheckBudget
  BEFORE UPDATE OF budget ON DEPARTMENT
  FOR EACH ROW
    WHEN ( NOT ( NEW.budget BETWEEN 1 AND 7000 ) )
      EXECUTE PROCEDURE AbortUpdate();
```

Implementation of an action

Action

```
CREATE OR REPLACE FUNCTION AbortUpdate()
RETURNS TRIGGER AS
$$
BEGIN
  RAISE EXCEPTION 'Budget of department % cannot be equal to %',
                  NEW.name, NEW.budget;
  RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

CREATE OR REPLACE TRIGGER statement

The results from processing when a trigger detects an incorrect value

A trigger detects an incorrect value

```
UPDATE DEPARTMENT
SET budget = 100000
WHERE name = 'Physics';
ERROR:  Budget of department Physics cannot be equal to 100000.0
CONTEXT:  PL/pgSQL function abortupdate() line 3 at RAISE;
```

The results from processing when a trigger does not detect an incorrect value

A trigger does not detect an incorrect value

```
UPDATE DEPARTMENT
SET budget = 100
WHERE name = 'Physics';
UPDATE 1
```

CREATE OR REPLACE TRIGGER statement

The following temporal options are available

- `BEFORE` - a trigger fires before a triggering event
- `AFTER` - a trigger fires after a triggering event
- `INSTEAD OF` - a trigger fires instead of a triggering event, it is typically used to correctly implement view update operation i.e. a correct modification of base relational tables through an update performed on a relational view

Sample applications of temporal options

Fire a trigger before `UPDATE` operationon a relational table `DEPARTMENT`

A sample temporal option

```
BEFORE UPDATE ON DEPARTMENT
```

Fire a trigger after `DELETE` operation on a relational table `COURSE`

A sample temporal option

```
AFTER DELETE ON COURSE
```

CREATE OR REPLACE TRIGGER statement

Fire a trigger instead of INSERT operation on a relational view EMPVIEW

```
A sample temporal option
INSTEAD OF INSERT ON EMPVIEW
```

Fire a trigger after INSERT into a table DEPARTMENT or after DELETE from a table DEPARTMENT

```
A sample temporal option
AFTER INSERT OR DELETE ON DEPARTMENT
```

Fire a trigger before UPDATE operation on a column budget in a relational table DEPARTMENT

```
A sample temporal option
BEFORE UPDATE OF budget ON DEPARTMENT
```

CREATE OR REPLACE TRIGGER statement

The following events can fire a trigger

- INSERT
- DELETE
- UPDATE
- TRUNCATE

Sample applications of DML events

A sample DML event

```
BEFORE UPDATE OF cnum, title ON COURSE
```

A sample DML event

```
AFTER INSERT ON DEPARTMENT
```

A sample DML event

```
INSTEAD OF TRUNCATE ON DEPARTMENT
```

A sample DML event

```
AFTER DELETE OR INSERT OR UPDATE ON COURSE
```

CREATE OR REPLACE TRIGGER statement

Condition determines whether a trigger processes its body after it has been fired

Sample applications of condition

```
WHEN (NEW.budget > 7000)                                    A sample condition
WHEN (OLD.budget < 1000 AND NEW.budget > 2000);             A sample condition
WHEN (NEW.amount > 1000 );                                  A sample condition
WHEN (OLD.credits IN (6, 12));                              A sample condition
```

OLD and NEW are record variable such that for

- INSERT triggering operation OLD contains no values and NEW contains the new values
- UPDATE triggering operation OLD contains the old values and NEW contains the new values
- DELETE triggering operation OLD contains the old values and NEW contains no values

Database Triggers

Outline

Statement database triggers

A statement trigger fires once either before or after a triggering event

A sample statement trigger records in a relational table `DELETE` and `INSERT` operations on a relational table

Trigger name

```
CREATE OR REPLACE TRIGGER Audit
```

Temporal option and event specification

```
  AFTER DELETE OR INSERT ON DEPARTMENT
```

Action

```
  FOR EACH STATEMENT
    EXECUTE FUNCTION InsertAudit();
```

A function `InsertAudit` inserts into a relational table `AUDIT` information about `DELETE` and `INSERT` operations together with the timestamps

Table AUDIT

```
CREATE TABLE AUDIT (
  statement VARCHAR(10) NOT NULL,
  date_time TIMESTAMP   NOT NULL,
    CONSTRAINT AUDIT_PK PRIMARY KEY (date_time) );
```

Statement database triggers

A variable `TG_OP` contains a name of operation that fired a trigger

A function InsertAudit

```
CREATE OR REPLACE FUNCTION InsertAudit()
RETURNS TRIGGER AS
$$
BEGIN
  INSERT INTO AUDIT VALUES(TG_OP, current_timestamp);
  RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Other special variables:

- `NEW`, `OLD`,
- `TG_NAME`: a name of trigger
- `TG_WHEN`: either `BEFORE` or `AFTER` or `INSTEAD`
- `TG_LEVEL`: either `ROW` or `STATEMENT` trigger

Statement database triggers

Testing with a single insertion, single deletion and many deletions

Testing

```
INSERT INTO DEPARTMENT VALUES ('Quantum Computers','QC',30,'Kate',223426.9 );
DELETE FROM DEPARTMENT WHERE name = 'Arts';
DELETE FROM DEPARTMENT;
```

Rows inserted into AUDIT table

```
SELECT * FROM AUDIT;
 statement |         date_time
-----------+----------------------------
 INSERT    | 2023-12-06 14:08:20.475765
 DELETE    | 2023-12-06 14:08:33.19901
 DELETE    | 2023-12-06 14:08:41.729328
```

Database Triggers

Outline

Row database triggers

A row trigger fires either after or before a triggering event affects a row in a relational table

- When a temporal option `BEFORE` is used a trigger fires once before a triggering event affects a row in a relational table
- When a temporal option `AFTER` is used a trigger fires once after a triggering event affects a row in a relational table

For example, if a temporal option and event are

```
BEFORE INSERT ON DEPARTMENT
```
A temporal option and event

then a trigger fires before each insertion into a relational table (it is possible to have many insertions when a multirow `INSERT` statement is processed)

For example, if a temporal option and event are

```
AFTER UPDATE ON EMPLOYEE
```
A temporal option and event

then a trigger fires after a row is updated in a relational table, if a triggering event updates n rows then a trigger fires n times

Row database triggers

For example, if a temporal option and event are

A temporal option and event

```
AFTER DELETE ON PROJECT
```

Then a trigger fires after a row is deleted from a relational table, if a triggering event deletes n rows then a trigger fires n times

Row database triggers

A sample row trigger that records in a relational table `AUDIT` information about each row updated in a relational table `DEPARTMENT`

```
CREATE OR REPLACE TRIGGER UpdateDepartment                    Trigger name
  AFTER UPDATE ON DEPARTMENT                                  Temporal option and event specification
  FOR EACH ROW                                                Action
    EXECUTE FUNCTION UpdateDepartment();
```

A function `UpdateDepartment` inserts into a relational table `AUDIT` information about `UPDATE` operation, timestamp, name of a department, old and new value of budget

```
                                                              Table AUDIT
CREATE TABLE AUDIT (
  statement VARCHAR(10)  NOT NULL,
  date_time TIMESTAMP    NOT NULL,
  name      VARCHAR(50)  NOT NULL,
  oldbudget DECIMAL(9,1) NOT NULL,
  newbudget DECIMAL(9,1) NOT NULL);
```

Row database triggers

A variable `TG_OP` contains a name of operation that fired a trigger

A variable `OLD.name` contains a name of departemnt

A variable `OLD.budget` contains a value of budget befor update and a variable `NEW.budget` contains a value of budget after update

A function UpdateDepartment

```
CREATE OR REPLACE FUNCTION UpdateDepartment()
RETURNS TRIGGER AS
$$
BEGIN
  INSERT INTO AUDIT VALUES(TG_OP, current_timestamp, OLD.name, OLD.budget,
                                                             NEW.budget);

  RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Statement database triggers

Testing with a single update of many rows in a relational table `DEPARTMENT`

Testing

```
UPDATE DEPARTMENT SET budget = budget + 1000;
UPDATE 7
```

Rows inserted into AUDIT table

```
SELECT * FROM AUDIT;
statement |         date_time          |       name         | oldbudget | newbudget
----------+----------------------------+--------------------+-----------+-----------
 UPDATE   | 2023-12-06 18:56:03.402062 | Computer Science   |  223426.9 |  224426.9
 UPDATE   | 2023-12-06 18:56:03.402062 | Physics            |  223453.9 |  224453.9
 UPDATE   | 2023-12-06 18:56:03.402062 | Engineering        |  313455.9 |  314455.9
 UPDATE   | 2023-12-06 18:56:03.402062 | Mathematics        |  133456.9 |  134456.9
 UPDATE   | 2023-12-06 18:56:03.402062 | Education          |  173557.9 |  174557.9
 UPDATE   | 2023-12-06 18:56:03.402062 | Science            |  163459.9 |  164459.9
 UPDATE   | 2023-12-06 18:56:03.402062 | Quantum Computers  |  223426.9 |  224426.9
(7 rows)
```

Database Triggers

Outline

Sample application

We consider a database created through processing of the following `CREATE TABLE` statements

Relational table DEPARTMENT

```
CREATE TABLE DEPARTMENT(
 name               VARCHAR(50)        NOT NULL,
 code               CHAR(5)            NOT NULL,
 total_staff_number DECIMAL(2)         NOT NULL,
 chair              VARCHAR(50)            NULL,
 budget             DECIMAL(9,1)       NOT NULL,
  CONSTRAINT dept_pkey PRIMARY KEY(name),
  CONSTRAINT dept_ckey1 UNIQUE(code),
  CONSTRAINT dept_ckey2 UNIQUE(chair),
  CONSTRAINT dept_check1 CHECK (total_staff_number BETWEEN 1 AND 50) );
```

Relational table COURSE

```
CREATE TABLE COURSE(
 cnum               CHAR(7)           NOT NULL,
 title              VARCHAR(200)      NOT NULL,
 credits            DECIMAL(2)        NOT NULL,
 offered_by         VARCHAR(50)           NULL,
  CONSTRAINT course_pkey PRIMARY KEY(cnum),
  CONSTRAINT course_check1 CHECK (credits IN (6, 12)),
  CONSTRAINT course_fkey1 FOREIGN KEY(offered_by)
                          REFERENCES DEPARTMENT(name) ON DELETE CASCADE );
```

Sample application

We would like to add information about the total number of courses taught by each department

A correct modification of a design adds a column `total_courses` to a relational table `DEPARTMENT`

Addition of a column `total_courses` does not violate BCNF for a relational table `DEPARTMENT`

There is no need to add any new relational table to the design

Adding a column total_courses to a table DEPARTMENT

```
ALTER TABLE DEPARTMENT ADD total_courses DECIMAL(2);
```

A column `total_courses` is filled with data in the following way

Filling a column total_courses with data

```
UPDATE DEPARTMENT
SET total_courses = (SELECT COUNT(*)
                     FROM COURSE
                     WHERE course.offered_by = DEPARTMENT.name);
```

Sample application

Each time a course is added or deleted or moved to another department a statement trigger supposed to update the values in a column `total_courses` in a relational table `DEPARTMENT`

```
                A statement trigger that updates a column total_courses in a table DEPARTMENT
CREATE OR REPLACE TRIGGER UpdateTotal
AFTER DELETE OR INSERT OR UPDATE ON COURSE
  FOR EACH STATEMENT
    EXECUTE FUNCTION ChangeTotal();
```

A function `ChangeTotal()` is implemented in the following way

```
                Updating the values in a column total_courses in a relational table DEPARTMENT
CREATE OR REPLACE FUNCTION ChangeTotal()
RETURNS TRIGGER AS
$$
BEGIN
  UPDATE DEPARTMENT
  SET total_courses = (SELECT COUNT(*)
                       FROM COURSE
                       WHERE course.offered_by = DEPARTMENT.name);
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Sample application

The same task can be implemented as a row trigger in the following way

A row trigger that updates a column total_courses in a table DEPARTMENT

```
CREATE OR REPLACE TRIGGER UpdateTotal
AFTER DELETE OR INSERT OR UPDATE OF offered_by ON COURSE
  FOR EACH ROW
    EXECUTE FUNCTION RowChangeTotal();
```

A function `RowChangeTotal()` is implemented in the following way

Updating the values in a column total_courses in a relational table DEPARTMENT

```
CREATE OR REPLACE FUNCTION RowChangeTotal()
RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'DELETE') THEN
    UPDATE DEPARTMENT SET total_courses = total_courses - 1 WHERE name = OLD.offered_by;
  ELSIF (TG_OP = 'INSERT') THEN
    UPDATE DEPARTMENT SET total_courses = total_courses + 1 WHERE name = NEW.offered_by;
  ELSE
    UPDATE DEPARTMENT SET total_courses = total_courses - 1 WHERE name = OLD.offered_by;
    UPDATE DEPARTMENT SET total_courses = total_courses + 1 WHERE name = NEW.offered_by;
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Database Triggers

Outline

Problems with database triggers

Infinite chains of trigger invocations

- What to do when a trigger `A` while processing its body fires a trigger `B` and a trigger `B` while processing its body fires a trigger `A` ?

Indeterministic trigger invocations

- It may happen that due to a database transaction serialization mechanisms the same chain of trigger invocations will be processed (serialized) in many different way by a transaction scheduler, e.g. if two triggers `A` and `B` fire in more or less the same moment in time then sometimes `A` will be processed before `B` and sometimes `B` will be processed before `A`

Lack of external control

- Long chains of trigger invocations contribute to very serious data security risks, e.g. it is possible to "hide" malicious code at the end of long chains of trigger invocations

Lack of design methodology

- The ad hoc uncontrolled and not well planned additions of new triggers lead to a situation where after addition or modification of a trigger there is no certainty that the chains of trigger invocations do not corrupt a databse

References

PostgreSQL 16.1 Documentation, Chapter 39. Triggers

PostgreSQL 16.1 Documentation, `CREATE TRIGGER`

PostgreSQL Tutorial, PostgreSQL Triggers

T. Connoly, C. Begg, Database Systems, A Practical Approach to Design, Implementation, and Management, Chapter 8 Advanced SQL, Pearson Education Ltd, 2015