Databases

TDA357/DIT621– LP3 2023

Lecture 8

Ana Bove

(much of the material is based on material from both Thomas Hallgren and Jonas Duregård)

February 6th 2023

Recall Last 2 Lectures

- Functional dependencies:
  - $X \to Y$: if two rows agree on the values of X then they must agree on the values of Y;
  - $R(S)$ in BCNF if for all non-trivial $X \to Y$ $(Y \not\subseteq X)$, X is a superkey;
  - Normalisation algorithm: given $R(S)$ and a BCNF-violation $X \to Y$, we split $R(S)$ into $R_1(X^+)$ and $R_2(X \cup (S - X^+))$ and recursively normalise.

- Multivalued functional dependencies:
  - $X \twoheadrightarrow Y$: if we have tuples $\langle \overline{x}, \overline{y_1}, \overline{z_1} \rangle$ and $\langle \overline{x}, \overline{y_2}, \overline{z_2} \rangle$, then we also have tuples $\langle \overline{x}, \overline{y_1}, \overline{z_2} \rangle$ and $\langle \overline{x}, \overline{y_2}, \overline{z_1} \rangle$;
  - $R(S)$ in 4NF if in BCNF and for all non-trivial $X \twoheadrightarrow Y$ $(Y \not\subseteq X$ or $S \not\subseteq X \cup Y)$, X is a superkey;
  - Normalisation algorithm: given $R(S)$ and a 4NF-violation $X \twoheadrightarrow Y$, we split $R(S)$ into $R_1(X \cup Y)$ and $R_2(S - Y)$ and recursively normalise.

Overview of Today's Lecture

- Functions;
- Triggers:
  - On tables;
  - On views;
- Example.

Function Definitions

Users can create functions to be used like other **SQL** functions (see documentation).

```
CREATE FUNCTION name (parameters) RETURNS type AS $$
   < function code in here >
$$ LANGUAGE language ;
```

```
CREATE OR REPLACE FUNCTION name (parameters) RETURNS type AS $$
   < function code in here >
$$ LANGUAGE language ;
```

Possible languages:

SQL:	pure **SQL** statements;
plpgsql:	PL/pgSQL - SQL Procedural Language (programs are sequence of instructions).

Example: Function Definitions

- Tables:

```
CREATE TABLE CourseRegistrations (
  student CHAR(10) REFERENCES ...,
  course CHAR(6) REFERENCES ...,
  position INT CHECK (position > 0),
  PRIMARY KEY (student, course) );
```

- Functions:

```
CREATE OR REPLACE FUNCTION nextPos (CHAR(6)) RETURNS INT AS $$
  SELECT COUNT(*) + 1 FROM CourseRegistrations WHERE course = $1
$$ LANGUAGE SQL;
```

- Using the functions:

```
INSERT INTO CourseRegistrations
VALUES ('1234567890', 'TDA357', nextPos ('TDA357'));

SELECT nextPos ('TDA357') AS NextPosition;

SELECT DISTINCT course, nextPos (course) FROM CourseRegistrations;
```

Example: Alternative Function Definition

PL/pgSQL is a procedural language and it gives us other possibilities when writing functions (more about functions in PL/pgSQL is coming!).

The function needs to end with a RETURN statement.

```
CREATE OR REPLACE FUNCTION nextPos (CHAR(6)) RETURNS INT AS $$
DECLARE
   pos INT;
BEGIN
   pos := (SELECT COUNT(*) FROM CourseRegistrations WHERE course = $1);
   RETURN (pos+1);
END
$$ LANGUAGE plpgsql;
```

The function is however used in the same way.

Variable Declaration and Assignment in PL/pgSQL

```
CREATE OR REPLACE FUNCTION my_function () RETURNS TRIGGER AS $$

DECLARE
   my_balance INT;
   total INT;
   my_text TEXT;
   ...

BEGIN
   ...
   SELECT balance INTO my_balance FROM Accounts WHERE id = '123456';
   ...
   total := (SELECT SUM(balance) FROM Accounts);
   ...
   my_text := 'Here I write my text';
   ...
END;

$$ LANGUAGE plpgsql;
```

Check documentation to see what happens if the query returns none / multiple rows: raises an error/ gives NULL/returns first row/...

Conditionals in PL/pgSQL

```
IF <condition> THEN
   ... ;
ELSEIF <condition> THEN
   ... ;
ELSE
   ... ;
END IF;
```

- ELSEIF and ELSE parts are optional;
- When the condition returns UNKNOWN, the ELSE part will be run;
- Useful condition to see if the query returns at least one element:

```
IF (EXISTS (SELECT ...)) THEN
   ... ;
... ;
```

Case Analysis in PL/pgSQL

```
CASE <expression>
  WHEN <value> THEN ... ;
  WHEN <value> THEN ... ;
  WHEN <value> THEN ... ;
  ... ;
  ELSE ... ;
END CASE;
```

```
CASE
  WHEN <condition> THEN ... ;
  WHEN <condition> THEN ... ;
  WHEN <condition> THEN ... ;
  ... ;
  ELSE ... ;
END CASE;
```

- At least one WHEN ... THEN ... is needed;
- ELSE is optional.

More PL/pgSQL Features

Loops:

```
LOOP EXIT WHEN <condition>;
  <code of the loop>
END LOOP;
```

Exceptions:

```
BEGIN
  <code that may cause an exception>
EXCEPTION
  WHEN <error code>
  <code that handles the exception>
END;
```

Check documention for error codes.

Using Functions in **SQL**

Using functions in this way when inserting values helps having the right position in a course but we are still not safe.

- It will work as expected only if the table is consistent from the start ...
- ... and when both codes match in the insert clause;
- Deleting a row will create "holes" and next time we use the function it might create collisions;
- One could use MAX instead of COUNT to avoid collisions ...
- ... but it is still possible to insert values without using the function;
- Recall assertions do not work in PostgeSQL!

Triggers

Triggers are procedures executed when certain actions take place (like updating, inserting or deleting from a table).

Triggers are useful when “something” is supposed to happen when “some other action” is performed, or when we need to ensure constraints/invariants across tables.

Example: If a student passed the missing pre-requisite for a certain course, then the student is not longer conditionally accepted to the course but gets a normal registration.

We then need to:

- Define the functions associated to the triggers;
- Specify which function the trigger calls and how.

All or Nothing Semantics

All **SQL** statements are atomic, even if they affect several rows: if there is an error no rows will be changed!

Constraints can prevent an INSERT or UPDATE from completing.

Also, intermediate changes are never visible to other users of the database.

Example: An electrical power problem in the middle of a money transaction should leave both accounts unchanged.

This includes trigger functions: if a trigger function fails, nothing is changed in the database!

(SQL statements can be grouped into larger transactions – will discuss them in a later lecture.)

Defining Functions to Be Used in Triggers

- The return type should then be TRIGGER and the language plpgsql;

```
CREATE FUNCTION name (parameters) RETURNS TRIGGER AS $$
   < function code in here >
$$ LANGUAGE plpgsql;
```

- The function needs to end with a RETURN statement (see slide 17);
- Raise an exception to prevent the trigger to do changes in the database if something is wrong (recall atomic actions!):

```
RAISE EXCEPTION 'simple error message';

RAISE EXCEPTION 'error message with more info %', error_text;
```

- Output message if needed (useful for debugging):

```
RAISE NOTICE 'simple info message';

RAISE NOTICE 'info message with more info %', error_text;
```

- See documentation on errors and messages in PostgeSQL manual.

Triggers on Tables

See documentation for further information on the creation of triggers.

```
CREATE TRIGGER trigger_name
  AFTER|BEFORE INSERT|UPDATE|DELETE ON table_name
  FOR EACH ROW|STATEMENT
  EXECUTE FUNCTION function_name ();
```

Can also combine events and have a condition when to be executed:

```
CREATE TRIGGER trigger_name
  AFTER UPDATE OR DELETE ON table_name
  FOR EACH ...
  WHEN condition
  EXECUTE FUNCTION function_name ();
```

Per-row vs. Per-statement Triggers

See documentation for more information on the behavior of triggers.

Per-row triggers: The function is invoked once for each row that is affected by the statement that fired the trigger (not just once for the action!);

Per-statement trigger: The function is invoked only once when an appropriate statement is executed, regardless of the number of rows affected (in particular, a statement that affects zero rows will still result in one execution).

Note: Per-statement triggers DO NOT have access to OLD!!!
So OLD.attribute will simple be NULL.

NEW and OLD

When executing a trigger function on a row/table entry:

- NEW.<attribute-name> refers to the new value of an attribute:
  - Can be used in functions triggered by INSERT or UPDATE;
  - It contains the value that has just been inserted/updated;
  - Possibly RETURN NEW in such functions.

- OLD.<attribute-name> refers to the old value of an attribute:
  - Can be used in functions triggered by DELETE or UPDATE;
  - It contains the value of the attribute in the row that has been (or is about to be) deleted/updated;
  - Possibly RETURN OLD in such functions.

Returning NEW, OLD or NULL?

Trigger functions shuould always end with a RETURN statement.
But what to return?

- For row level triggers, a row of the table on which the trigger is defined (NEW or OLD);
- For statement level triggers, the value NULL.

Observe that:

- The return value is ignored for row level AFTER triggers;
- In row level BEFORE triggers:
  - If the trigger returns NULL, the triggering operation is aborted, and the row will not be modified!
  - For INSERT and UPDATE triggers, the returned row is the input for the triggering statement.

Read this for more information.

Example: A Trigger for Transferences between Accounts

```
CREATE OR REPLACE FUNCTION make_transfer () RETURNS TRIGGER AS $$
BEGIN
  UPDATE Accounts SET balance = balance - NEW.amount
  WHERE id=NEW.sender;
  UPDATE Accounts SET balance = balance + NEW.amount
  WHERE id=NEW.receiver;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;


 DROP TRIGGER IF EXISTS after_insert_on_transfer ON Transfers;


CREATE TRIGGER after_insert_on_transfer
  AFTER INSERT ON Transfers
  FOR EACH ROW
  EXECUTE FUNCTION make_transfer ();
```

Note: If we set BEFORE in the trigger, it will still do the work.

Example: A Trigger to Guarantee a Minimum Balance

```
CREATE FUNCTION checkMinimumTotalBalance () RETURNS TRIGGER AS $$
DECLARE
  total INT;
BEGIN
  total := (SELECT SUM(balance) FROM Accounts);
  IF total < 30000 THEN
    RAISE EXCEPTION 'error: total balance would be %, which is too low', total;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;


 CREATE TRIGGER minimumBankBalance
  AFTER UPDATE OR DELETE ON Accounts
  FOR EACH STATEMENT
  EXECUTE FUNCTION checkMinimumTotalBalance ();
```

Note: If we set BEFORE in the trigger, then the check will be done before the update/deletion and will not make sense!

Trigges on Views

Views are used for querying but not for modifying data.

However, with an INSTEAD OF trigger we can make changes to the underlying tables in response to INSERT, UPDATE or DELETE on the view.

```
CREATE TRIGGER trigger_name
  INSTEAD OF INSERT|UPDATE|DELETE ON view_name
  FOR EACH ROW|STATEMENT
  EXECUTE FUNCTION function_name ();
```

If for example the trigger is INSTEAD OF INSERT, we can then insert on/via the view:

```
INSERT INTO view_name VALUES (...);
```

Using Functions in Triggers

- If the function *raises an error in any of the rows* that are affected, the *whole* transaction is "rolled back" and the database is not modified at all!
- One needs to be careful not to have an infinite recursive function call!
  Example: An UPDATE trigger that performs UPDATEs!

  One could avoid infinite recursive calls with the following condition:

  ```
  WHEN (pg_trigger_depth() < 1)
  ```
- When a trigger reports "X rows deleted" it actually means that the trigger was executed for X rows, the database may not have been changed at all;
- Triggers on tables or on views? Depends on what you are doing but views are the interface of the database and it is usually safe to work on them!

Constraints, Views and Triggers

Database integrity can be improved by several techniques:

SQL Constraints: conditions on attribute values and tuples.

Views: virtual tables that show useful information that would create redundancy if stored in the actual tables.

Triggers (and assertions if available): automated checks and actions performed on entire/several tables.

As a general rule, these methods should be applied in the above order: if a constraint or a view can adequately do the job, do not use a trigger!

Example: Constraints, Views and Triggers (I)

You need to implement a database with this description, but you are allowed to divide it across as many tables and views as you need to.

Assignments (<u>course</u>, <u>name</u>, description, deadline)
Submissions (<u>idnr</u>, student, course, assignment, stime)
  (course, assignment) $\rightarrow$ Assignments.(course, name)
SubmittedFiles (<u>submission</u>, <u>filename</u>, filesize, contents)
  submission $\rightarrow$ Submissions.idnr
Registered (<u>student</u>, <u>course</u>)

You should also enforce the following additional constraints:

1. The newer a submission is, the higher its idnr;
2. Students can only submit solutions to assignments in courses they are registered for;
3. filesize always reflects the length of content (computable by length(contents) in PostgreSQL); a database user should not need to specify filesize when adding a new file to a submission;
4. When a submission is deleted, its files should also be deleted automatically;
5. When all files of a submission are deleted, delete the submission itself as well;
6. A student cannot have two submissions for the same assignment with exactly the same time.

Example: Constraints, Views and Triggers (I, Cont.)

For the **SQL** code with the tables and views see the end of the file lecture8.sql (available from Canvas).

1. A trigger on INSERT OR UPDATE on Submissions that either automatically adjusts the idnr or rejects updates that violate the rule;
2. See reference to Registrations in Submissions;
3. See view SubmittedFiles;
4. See ON DELETE CASCADE on the reference to Submissions in SubmittedFileTables;
5. A trigger AFTER DELETE (and maybe even OR UPDATE to be safe) for SubmittedFiles that checks if the last file was deleted and if so, run DELETE FROM Submissions WHERE idnr = OLD.submission;
6. See UNIQUE constraint in Submissions.

Example: Constraints, Views and Triggers (II)

Consider a database for storing messages with at least the following interface (there could be more tables/views):

Messages (<u>id</u>, sender, receiver, text, time)
RemovedMessages (<u>id</u>, sender, receiver, text, time)

No message should appear in both of these tables/views.

1. Explain how the database could be implemented so that one can remove a message with a single **SQL** statement; avoid using triggers if possible;
2. Explain also how to keep at most the latest 100 messages removed from each message receiver, permanently deleting any older messages from the database.

Example: Constraints, Views and Triggers (II, Cont.)

1. Some possible solutions that have both Messages and RemovedMessages as views:
   1. Have a single table, like Messages but with an extra attribute isDeleted of type BOOLEAN; then the views simply filter on that condition or its negation.
      The removal operation just sets isDeleted to TRUE.
   2. Have a table for all messages, and one for the ids of removed messages (with a reference to the other table). Messages view uses NOT IN and RemovedMessages is just a join on the two tables.
      The removal operation is an insert into the table for removed messages.
   3. Like above but with a table for the messages (ids) in the "inbox".
2. Depending on which of the approaches above you use, a trigger after UPDATE, INSERT or DELETE could be used.
   If using a FOR EACH ROW trigger, it should be enough to use an IF-statement that checks if there are more than 100 messages for the same receiver (as OLD or NEW) and delete the oldest row for that receiver if there is.

Overview of Next Lecture

- Databases in software application (by Jonas).

Reading:

Book:	chapter 9
Notes:	chapter 8