Advanced Management of Data
Exercise 7 Topic 2:
Extensions of SQL

Triggers
More academic examples

- views are nice, but one could achieve the same just with triggers
- disclaimer: you shouldn't do this in real world scenarios, but as this is an academic example…
  - if you don't like academic examples you could easily exchange the simple logic in the following task with some complex system of enterprise logic, that heavily depends on triggers to enforce all constraints between different relations
    - by the way, this is exactly what you should be able to do after heaving learned some of the possibilities of PL/pgSQL in this exercise, but we won't practice this for practical reasons (we don't have enough time and you won't learn this much, but you could do it as homework if you like)
- create a new table, that stores just one value in a column, which is the sum of all numbers in the table `numbers`

```
CREATE TABLE IF NOT EXISTS numbers_sum (number_sum INTEGER);

INSERT INTO numbers_sum SELECT sum(number) FROM numbers; -- initialize once
```

- <u>Task</u>: this sum should be updated, if `numbers` is changed and trying to change this table `numbers_sum` should result in changes to the table `numbers`

Triggers
Update on change

SAMPLE CODE GOT PROVIDED DURING THE EXERCISE

Triggers
Prevent INSERT and DELETE but propagate them to numbers

SAMPLE CODE GOT PROVIDED DURING THE EXERCISE

Triggers
Propagate TRUNCATE and restore row

SAMPLE CODE GOT PROVIDED DURING THE EXERCISE

- now we manage INSERT, DELETE and TRUNCATE, but what about modifying this sum on our own?

Triggers
Just don’t let it happen

```
SAMPLE CODE GOT PROVIDED DURING THE EXERCISE
```

- we also replace the update with the correct update function, but only do it once to avoid recursion
- but don’t try to propagate updates to numbers_sum back to numbers
    - it would be possible to propagate updates just in the case, where the new value isn’t the sum of all numbers, but this will certainly lead to race conditions

Triggers
Some last words

- creating many different triggers on one relation can lead to confusion, what is going on
- especially triggers that abort execution of all further triggers can easily prevent your precious enterprise logic from working
- just remember that in PostgreSQL all triggers are executed in alphabetical order
- so you can just disable most triggers with an 'aaa'-trigger, that prevents further execution
- or you simply could do something like

```
    ALTER TABLE tablename DISABLE TRIGGER triggername;
```

- and re-enable them with

```
    ALTER TABLE tablename ENABLE TRIGGER triggername;
```

Recursive Queries
Common Table Expressions (CTE)

- recursive queries are useful for queries that need to refer to their own output
- for example this is the case for hierarchical data
- a recursive query consists of three parts
  - a non-recursice term forms the base result set
  - a `UNION` or `UNION ALL`
  - a recursive term containing a reference to the queries own output

```
WITH RECURSIVE tabledefintion AS (
  non-recursive term
  UNION [ALL]
  recursive term
)
SELECT whatever FROM tabledefintion;
```

Recursive Queries
Very small example

- a small example might help to understand it

```
WITH RECURSIVE numbers (number) AS (
    VALUES (1)
  UNION ALL
    SELECT number + 1 FROM numbers WHERE number < 100
)
SELECT sum(number) FROM numbers;
```

- this just sums the integers from 1 through 100

Recursive Queries
Typical graph

- lets give it a try and build a graph
- the graph is built from links between nodes and there must not be a link to the same node

```
CREATE TABLE graph (
  id SERIAL PRIMARY KEY,
  parentid INT REFERENCES graph(id),
  CONSTRAINT no_self_link CHECK (id <> parentid)
);
```

- Task: add some logic to prevent cycles in `graph`

Recursive Queries
Cycle-free graph

```
CREATE OR REPLACE FUNCTION detect_cycle() RETURNS TRIGGER AS $$
  BEGIN
    IF EXISTS (
      WITH RECURSIVE search_graph (parentid, path, cycle) AS (
        SELECT NEW.parentid, ARRAY[NEW.id, NEW.parentid], (NEW.id = NEW.parentid)
          FROM graph WHERE id = NEW.parentid
      UNION ALL
        SELECT graph.parentid, path || graph.parentid, graph.parentid = ANY(path)
          FROM search_graph JOIN graph ON id = search_graph.parentid WHERE NOT cycle
      )
      SELECT 1 FROM search_graph WHERE cycle LIMIT 1
    ) THEN
      RETURN NULL;
    END IF;
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

Recursive Queries
Cycle-free graph

```
CREATE TRIGGER prevent_cycle BEFORE INSERT OR UPDATE ON graph
  FOR EACH ROW
    EXECUTE PROCEDURE detect_cycle();
```

- starting from the new `id` we check if there is any ancestor with the same `id` and stop if we found one
- the search is done by building an array with all ancestors