Advanced Management of Data
Exercise 4 Topic 2:
Extensions of SQL

Exceptions

- at this time our function `random42()` returns numbers in a range of 1 to 100
- Task: remove the usage of `ceil()` so that it returns numbers in a range of 0 to 100
  - further modify the function to raise the exception `numeric_value_out_of_range` when it dices a “0”

Exceptions
Throw

```
CREATE OR REPLACE FUNCTION random42() RETURNS SETOF INTEGER AS $$
  DECLARE
    rnd INTEGER;
  BEGIN
    LOOP
      rnd = (random() * 100)::INTEGER;
      IF rnd = 0 THEN
        RAISE numeric_value_out_of_range;
      END IF;
      RETURN NEXT rnd;
      EXIT WHEN rnd = 42;
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

- as `random42()` might throw an exception, we have to catch it in `count42()`
- Task: modify it to return “0” if this exception occurs

Exceptions
Catch

```
CREATE OR REPLACE FUNCTION count42() RETURNS INTEGER AS $$
  DECLARE
    c INTEGER = 1;
  BEGIN
    WHILE (SELECT COUNT(*) FROM random42()) != 42 LOOP
      c = c + 1;
    END LOOP;
    RETURN c;
    EXCEPTION
      WHEN numeric_value_out_of_range THEN
        RETURN 0;
  END;
$$ LANGUAGE plpgsql;
```

- as `random42()` now throws an exception in about every second run, you almost never see `count42()` returning anything but “0”
- Task: to make things more interesting again and test whether casting `FLOAT` to `INTEGER` is really rounding, modify `random42()` to throw another exception `error_in_assignment`, when it dices a “100”

Exceptions
Throw once more

```
CREATE OR REPLACE FUNCTION random42() RETURNS SETOF INTEGER AS $$
  DECLARE
    rnd INTEGER;
  BEGIN
    LOOP
      rnd = (random() * 100)::INTEGER;
      IF rnd = 0 THEN
        RAISE numeric_value_out_of_range;
      ELSEIF rnd = 100 THEN
        RAISE error_in_assignment;
      END IF;
      RETURN NEXT rnd;
      EXIT WHEN rnd = 42;
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

- of course we have to modify `count42()` now to catch this new exception
- Task: change it to return “-1” if this new exception occurs

Exceptions
Catch once more

```
CREATE OR REPLACE FUNCTION count42() RETURNS INTEGER AS $$
  DECLARE
    c INTEGER = 1;
  BEGIN
    WHILE (SELECT COUNT(*) FROM random42()) != 42 LOOP
      c = c + 1;
    END LOOP;
    RETURN c;
    EXCEPTION
      WHEN numeric_value_out_of_range THEN
        RETURN 0;
      WHEN error_in_assignment THEN
        RETURN -1;
  END;
$$ LANGUAGE plpgsql;
```

- now `count42()` should evenly return “0” and “-1”
- there are many more possible error codes that can be raised or caught
  - see https://www.postgresql.org/docs/current/static/errcodes-appendix.html

Cursors

- Cursors are useful for
  - processing result sets of queries with more than one row,
  - avoiding memory overrun when result set contains a large number of rows and
  - returning large result sets in an efficient way from one function, as it just needs to return a cursor to a result set
- the simplest way of using cursors is a `FOR` loop, as it uses a cursor internally

```
DECLARE
  target SOMETYPE;
BEGIN
  FOR target IN query LOOP
    statements
  END LOOP;
END;
```

Cursors

- achieving the same with cursors is much more code

```
DECLARE
  cursorname CURSOR FOR query;
  target SOMETYPE;
BEGIN
  OPEN cursorname;
  LOOP
    FETCH cursorname INTO target; -- FETCH NEXT IN cursorname INTO target;
    EXIT WHEN NOT FOUND; -- target is NULL if there are no more rows found
    statements
  END LOOP;
  CLOSE cursorname; -- be nice and close it afterwards
END;
```

Cursors

- fortunately there is also a `FOR` construct for bound cursors, but they must not be open at this point, as they are opened in before and closed afterwards by this construct

```
DECLARE
  cursorname CURSOR FOR query;
  target SOMETYPE;
BEGIN
  FOR target IN cursorname LOOP
    statements
  END LOOP;
END;
```

Cursors

- it is also possible to declare unbound cursors, that are bound later to any query at opening time

```
DECLARE
  cursorname REFCURSOR; -- all cursor variables have the type REFCURSOR
BEGIN
  OPEN cursorname FOR query;
```

- there are more options for cursors and some other syntaxes
  - see https://www.postgresql.org/docs/current/static/plpgsql-cursors.html

Cursors

- unfortunately there don't exist the useful cursor attributes %FOUND, %NOTFOUND, %ISOPEN and %ROWCOUNT like in Oracle's PL/SQL
- but there is a local variable FOUND, that has the status of the last query
- and you can use a diagnostic query concerning the number of rows affected by the last query

```
GET DIAGNOSTICS rowcount = ROW_COUNT;
```

- if you like to count all rows under the cursor, you just have to use a query affecting all rows

Cursors
Wrapping around

- now its time to do something useful with cursors
- we have our nice `random42()` function, that is returning a bunch of numbers from 1 to 99 or raising one of two exceptions when 0 or 100 occurs before 42
- Task: write some kind of wrapper function for this, that is returning a cursor to an instance without an exception

Cursors
Wrapping around

```
CREATE OR REPLACE FUNCTION wrap_random42() RETURNS REFCURSOR AS $$
  DECLARE
    curs CURSOR FOR SELECT * FROM random42();
  BEGIN
    LOOP                          -- endless loop
      BEGIN                       -- new block needed for exception
        OPEN curs;                -- open cursor
        MOVE FORWARD FROM curs;   -- try to trigger an error
        MOVE BACKWARD IN curs;    -- rewind cursor – both FROM curs and IN curs are possible
        RETURN curs;              -- return cursor
      EXCEPTION                   -- exceptions have to be at the end of a block
        WHEN OTHERS THEN          -- catch any exception, ignore it and try again
          NULL;                   -- don't try to close the cursor here, as it is not open
      END;
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

Cursors
Unwrapping it

- Task: write another function that uses this new wrapper function and
  - multiplies the values of the first and the last row and
  - divides this by the number of rows and
  - returns this as `REAL` value (should be somewhere between 0.1 and 2079.0)

Cursors
Unwrapping it

```
CREATE OR REPLACE FUNCTION unwrap_random42() RETURNS REAL AS $$
  DECLARE
    curs REFCURSOR;
    firstrow REAL;
    lastrow REAL;
    rows REAL;
  BEGIN
    curs = wrap_random42();              -- get already opened cursor
    FETCH FIRST IN curs INTO firstrow;   -- read first row
    FETCH LAST FROM curs INTO lastrow;   -- read last row
    MOVE FORWARD FROM curs;              -- move to the end
    MOVE BACKWARD ALL IN curs;           -- go back all the way, so that we can count rows
    GET DIAGNOSTICS rows = ROW_COUNT;    -- get number of rows
    CLOSE curs;                          -- close cursor
    RETURN firstrow * lastrow / rows;    -- calculate and return
  END;
$$ LANGUAGE plpgsql;
```

Packages

- unlike Oracle's PL/SQL PostgreSQL's PL/pgSQL doesn't have packages
- it's advised to use schemas instead to organize functions into groups
- of course there are no package-level variables either, but one could keep per-session state in temporary tables instead